

DTIC COPY

2

AVF Control Number: AVF-VSR-338.0290  
89-08-15-HPC

AD-A221 652

Ada COMPILER  
VALIDATION SUMMARY REPORT:  
Certificate Number: 891019W1.10178  
Hewlett Packard Company  
HP 9000 Series 800 Ada Compiler, Version 4.35  
HP 9000 Series 800 Model 850

Completion of On-Site Testing:  
19 October 1989

Prepared By:  
Ada Validation Facility  
ASD/SCOL  
Wright-Patterson AFB OH 45433-6503

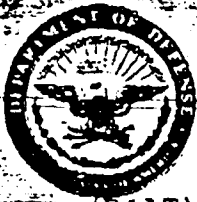
Prepared For:  
Ada Joint Program Office  
United States Department of Defense  
Washington DC 20301-3081

DTIC  
ELECTE  
MAY 17 1990  
S B D

DISTRIBUTION STATEMENT A

Approved for public release;  
Distribution Unlimited

90 05 11 022



(R&AT)

OFFICE OF THE DIRECTOR OF  
DEFENSE RESEARCH AND ENGINEERING

WASHINGTON, DC 20301

10 APR 1990

MEMORANDUM FOR Director, Directorate of Database Services,  
Defense Logistics Agency

SUBJECT: Technology Screening of Unclassified/Unlimited Reports

Your letter of 2 February 1990 to the Commander, Air Force Systems Command, Air Force Aeronautical Laboratory, Wright-Patterson Air Force Base stated that the Ada Validation Summary report for Meridian Software Systems, Inc. contained technical data that should be denied public disclosure according to DoD Directive 5230.25

We do not agree with this opinion that the contents of this particular Ada Validation Summary Report or the contents of the several hundred of such reports produced each year to document the conformity testing results of Ada compilers. Ada is not used exclusively for military applications. The language is an ANSI Military Standard, a Federal Information Processing Standard, and an International Standards Organization standard. Compilers are tested for conformity to the standard as the basis for obtaining an Ada Joint Program Office certificate of conformity. The results of this testing are documented in a standard form in all Ada Validation Summary Reports which the compiler vendor agrees to make public as part of his contract with the testing facility.

On 18 December 1985, the Commerce Department issued Part 379 Technical Data of the Export Administration specifically listing Ada Programming Support Environments (including compilers) as items controlled by the Commerce Department. The AJPO complies with Department of Commerce export control regulations. When Defense Technical Information Center receives an Ada Validation Summary Report, which may be produced by any of the five U.S. and European Ada Validation Facilities, the content should be made available to the public.

If you have any further questions, please feel free to contact the undersigned at (202) 694-0209.

*John P. Solomond*

John P. Solomond  
Director  
Ada Joint Program Office

**UNCLASSIFIED**

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS OF THIS COMPLETION FORM
1. REPORT NUMBER	12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) <b>Ada Compiler Validation Summary Report: Hewlett Packard Company, HP 9000 Series 800 Ada Compiler, Version 4.35 HP 9000 Series 800 Model 850 (Host &amp; Target), 891019W1</b>		5. TYPE OF REPORT & PERIOD COVERED 19 Oct. 1989 to 19 Oct. 1990
7. AUTHOR(s) <b>Wright-Patterson AFB Dayton, OH, USA</b>		6. PERFORMING ORG. REPORT NUMBER 10178
8. PERFORMING ORGANIZATION AND ADDRESS <b>Wright-Patterson AFB Dayton, OH, USA</b>		9. CONTRACT OR GRANT NUMBER(s)
11. CONTROLLING OFFICE NAME AND ADDRESS <b>Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081</b>		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) <b>Wright-Patterson AFB Dayton, OH, USA</b>		12. REPORT DATE
		13. NUMBER OF PAGES
		15. SECURITY CLASS (of this report) <b>UNCLASSIFIED</b>
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE <b>N/A</b>
16. DISTRIBUTION STATEMENT (of this Report) <b>Approved for public release; distribution unlimited.</b>		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 if different from Report) <b>UNCLASSIFIED</b>		
18. SUPPLEMENTARY NOTES		
19. KEYWORDS (Continue on reverse side if necessary and identify by block number) <b>Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO</b>		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) <b>Hewlett Packard Company, HP 9000 Series 800 Ada Compiler, Version 4.35, Wright-Patterson AFB, HP 9000 Series 800 Model 850 under HP-UX, Version A.B3.10 (release 3.1), ACVC 1.10.</b>		

DD FORM 1473  
1 JAN 73EDITION OF 1 NOV 65 IS OBSOLETE  
S/N 8102-LF-014-8801**UNCLASSIFIED**

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Ada Compiler Validation Summary Report:

Compiler Name: HP 9000 Series 800 Ada Compiler, Version 4.35

Certificate Number: 891019W1.10178


Host: HP 9000 Series 800 Model 850 under  
HP-UX, Version A.B3.10 (release 3.1)


Target: HP 9000 Series 800 Model 850 under  
HP-UX, Version A.B3.10 (release 3.1)

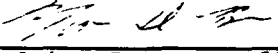
Testing Completed 19 October 1989 Using ACVC 1.10

Customer Agreement Number: 89-08-15-HPC

This report has been reviewed and is approved.

  
\_\_\_\_\_  
Ada Validation Facility  
Steven P. Wilson  
Technical Director  
ASD/SCOL  
Wright-Patterson AFB OH 45433-6503

  
\_\_\_\_\_  
for Ada Validation Organization  
Director, Computer & Software Engineering Division  
Institute for Defense Analyses  
Alexandria VA 22311

  
\_\_\_\_\_  
for Ada Joint Program Office  
Dr. John Solomond  
Director  
Department of Defense  
Washington DC 20301

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

## TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT . . . .	1-2
1.2	USE OF THIS VALIDATION SUMMARY REPORT . . . . .	1-2
1.3	REFERENCES. . . . .	1-3
1.4	DEFINITION OF TERMS . . . . .	1-3
1.5	ACVC TEST CLASSES . . . . .	1-4
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED. . . . .	2-1
2.2	IMPLEMENTATION CHARACTERISTICS. . . . .	2-2
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS. . . . .	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS. . . . .	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER. . . . .	3-2
3.4	WITHDRAWN TESTS . . . . .	3-2
3.5	INAPPLICABLE TESTS. . . . .	3-2
3.6	TEST, PROCESSING, AND EVALUATION MODIFICATIONS. .	3-6
3.7	ADDITIONAL TESTING INFORMATION. . . . .	3-7
3.7.1	Prevalidation . . . . .	3-7
3.7.2	Test Method . . . . .	3-7
3.7.3	Test Site . . . . .	3-8
APPENDIX A	DECLARATION OF CONFORMANCE	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	
APPENDIX E	COMPILER OPTIONS AS SUPPLIED BY THE HEWLETT PACKARD COMPANY	

## CHAPTER 1

### INTRODUCTION

This Validation Summary Report ~~(VSR)~~ describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability, ~~(ACVC)~~. An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

(2R)

## INTRODUCTION

### 1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- . To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 19 October 1989 at Cupertino CA.

### 1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C.#552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse  
Ada Joint Program Office  
OUSDRE  
The Pentagon, Rm 3D-139 (Fern Street)  
Washington DC 20301-3081

or from:

Ada Validation Facility  
ASD/SCOL  
Wright-Patterson AFB OH 45433-6503

## INTRODUCTION

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization  
Institute for Defense Analyses  
1801 North Beauregard Street  
Alexandria VA 22311

### 1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program Office, May 1989.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

### 1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the <u>Ada Compiler Validation Procedures</u> .
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.



## INTRODUCTION

Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

### 1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation of legal Ada programs with certain language constructs which cannot be verified at compile time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

## INTRODUCTION

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK\_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK\_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK\_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be

## INTRODUCTION

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

## CHAPTER 2

### CONFIGURATION INFORMATION

#### 2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: HP 9000 Series 800 Ada Compiler, Version 4.35

ACVC Version: 1.10

Certificate Number: 891019W1.10178

Host Computer:

Machine: HP 9000 Series 800 Model 850

Operating System: HP-UX  
Version A.B3.10 (release 3.1)

Memory Size: 96 Mb

Target Computer:

Machine: HP 9000 Series 800 Model 850

Operating System: HP-UX  
Version A.B3.10 (release 3.1)

Memory Size: 26 Mb

## CONFIGURATION INFORMATION

### 2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

#### a. Capacities.

- (1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)
- (2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)
- (3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)
- (4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

#### b. Predefined types.

- (1) This implementation supports the additional predefined types `SHORT_INTEGER`, `SHORT_SHORT_INTEGER`, and `LONG_FLOAT` in package `STANDARD`. (See tests B86001T..Z (7 tests).)

#### c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

- (1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)
- (2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)
- (3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

## CONFIGURATION INFORMATION

- (4) `NUMERIC_ERROR` is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)
- (5) Sometimes `NUMERIC_ERROR` is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)
- (6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

### d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

- (1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)
- (2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)
- (3) The method used for rounding to integer in static universal real expressions is round to even. (See test C4A014A.)

### e. Array types.

An implementation is allowed to raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` for an array having a `'LENGTH` that exceeds `STANDARD.INTEGER'LAST` and/or `SYSTEM.MAX_INT`.

For this implementation:

- (1) Declaration of an array type or subtype declaration with more than `SYSTEM.MAX_INT` components raises `NUMERIC_ERROR`. (See test C36003A.)
- (2) `NUMERIC_ERROR` is raised when an array type with `INTEGER'LAST + 2` components with each component being a null array is declared. (See test C36202A.)
- (3) `NUMERIC_ERROR` is raised when an array type with `SYSTEM.MAX_INT + 2` components with each component being a null array is declared. (See test C36202B.)
- (4) A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` raises `NUMERIC_ERROR` when the array type is declared. (See test C52103X.)

## CONFIGURATION INFORMATION

- (5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC\_ERROR when the array type is declared. (See test C52104Y.)
- (6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC\_ERROR or CONSTRAINT\_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC\_ERROR when the array type is declared. (See test E52103Y.)
- (7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT\_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)
- (8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT\_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

### f. Discriminated types.

- (1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT\_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

### g. Aggregates.

- (1) In the evaluation of a multi-dimensional aggregate, all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)
- (2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)
- (3) CONSTRAINT\_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

### h. Pragmas.

- (1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

## CONFIGURATION INFORMATION

### i. Generics.

- (1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)
- (2) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

### j. Input and output.

- (1) The package SEQUENTIAL\_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)
- (2) The package DIRECT\_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)
- (3) Modes IN FILE and OUT FILE are supported for SEQUENTIAL\_IO. (See tests CE2102D..E (2 tests), CE2102N, and CE2102P.)
- (4) Modes IN FILE, OUT FILE, and INOUT FILE are supported for DIRECT\_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)
- (5) Modes IN FILE and OUT FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)
- (6) RESET and DELETE operations are supported for SEQUENTIAL\_IO. (See tests CE2102G and CE2102X.)
- (7) RESET and DELETE operations are supported for DIRECT\_IO. (See tests CE2102K and CE2102Y.)
- (8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)
- (9) Overwriting to a sequential file truncates to the last element written. (See test CE2208B.)
- (10) Temporary sequential files are given names and deleted when closed. (See test CE2108A.)
- (11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)
- (12) Temporary text files are given names and deleted when closed. (See test CE3112A.)



## CONFIGURATION INFORMATION

- (13) More than one internal file can be associated with each external file for sequential files when writing or reading. (See tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)
- (14) More than one internal file can be associated with each external file for direct files when writing or reading. (See tests CE2107F..H (3 tests), CE2110D, and CE2111H.)
- (15) More than one internal file can be associated with each external file for text files when writing or reading. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

CHAPTER 3  
TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 362 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 35 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1132	1961	17	26	46	3311
Inapplicable	0	6	354	0	2	0	362
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

## TEST INFORMATION

### 3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER														TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14		
Passed	198	577	545	245	172	99	160	332	137	36	252	261	297	3311	
Inappl	14	72	135	3	0	0	6	0	0	0	0	108	24	362	
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44	
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717	

### 3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84M	CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110
ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

### 3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 362 tests were inapplicable for the reasons indicated:

- a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113L..Y (14 tests) C35705L..Y (14 tests) C35706L..Y (14 tests)  
 C35707L..Y (14 tests) C35708L..Y (14 tests) C35802L..Z (15 tests)  
 C45241L..Y (14 tests) C45321L..Y (14 tests) C45421L..Y (14 tests)

# TEST INFORMATION

C45521L..Z (15 tests) C45524L..Z (15 tests) C45621L..Z (15 tests)  
C45641L..Y (14 tests) C46012L..Z (15 tests)

- b. C35702A and B86001T are not applicable because this implementation supports no predefined type `SHORT_FLOAT`.
- c. The following 16 tests are not applicable because this implementation does not support a predefined type `LONG_INTEGER`:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

- d. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of `SYSTEM.MAX_MANTISSA` is less than 48.
- e. C86001F is not applicable because, for this implementation, the package `TEXT_IO` is dependent upon package `SYSTEM`. This test recompiles package `SYSTEM`, making package `TEXT_IO`, and hence package `REPORT`, obsolete.
- f. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than `DURATION`.
- g. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`.
- h. C87B62B applies the attribute `'STORAGE_SIZE` to an access type for which no `STORAGE_SIZE` length clause is given, raising `STORAGE_ERROR`. The AVO ruled that this behavior is acceptable and the test is not applicable.
- i. CD1009C, CD2A41A..B (2 tests), CD2A41E, and CD2A42A..J (10 tests) are not applicable because this implementation does not support size clauses for floating point types.
- j. The following 24 tests are not applicable because this implementation does not support size clauses for derived private types:

CD1C04A	CD2A21C	CD2A21D	CD2A22C	CD2A22D
CD2A22G	CD2A22H	CD2A31C	CD2A31D	CD2A32C
CD2A32D	CD2A32G	CD2A32H	CD2A41C	CD2A41D
CD2A51C	CD2A51D	CD2A52C	CD2A52D	CD2A52G
CD2A52H	CD2A53D	CD2A54D	CD2A54H	

- k. CD1C04B, CD1C04E, and CD4051A..D (4 tests) are not applicable because this implementation does not support representation clauses on derived records or derived tasks.

# TEST INFORMATION

- l. CD2A61A..D (4 tests), CD2A61F, CD2A61H..L (5 tests), CD2A62A..C (3 tests), CD2A71A..D (4 tests), CD2A72A..D (4 tests), CD2A74A..D (4 tests), and CD2A75A..D (4 tests) are not applicable because of the way this implementation allocates storage space for one component of an array or a record. The size specification clause for an array type or for a record type requires compression of the storage space needed for all the components, without gaps.

- m. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because this implementation does not support size clauses for access types.

- n. The following 21 tests are not applicable because this implementation does not support an address clause for a constant:

CD5011B	CD5011D	CD5011F	CD5011H	CD5011L
CD5011N	CD5011R	CD5012C	CD5012D	CD5012G
CD5012H	CD5012L	CD5013B	CD5013D	CD5013F
CD5013H	CD5013L	CD5013N	CD5013R	CD5014U
CD5014W				

- o. CD5012J, CD5013S, and CD5014S are not applicable because this implementation does not support address clauses for tasks.

- p. CE2102D is inapplicable because this implementation supports CREATE with IN\_FILE mode for SEQUENTIAL\_IO.

- q. CE2102E is inapplicable because this implementation supports CREATE with OUT\_FILE mode for SEQUENTIAL\_IO.

- r. CE2102F is inapplicable because this implementation supports CREATE with INOUT\_FILE mode for DIRECT\_IO.

- s. CE2102I is inapplicable because this implementation supports CREATE with IN\_FILE mode for DIRECT\_IO.

- t. CE2102J is inapplicable because this implementation supports CREATE with OUT\_FILE mode for DIRECT\_IO.

- u. CE2102N is inapplicable because this implementation supports OPEN with IN\_FILE mode for SEQUENTIAL\_IO.

- v. CE2102O is inapplicable because this implementation supports RESET with IN\_FILE mode for SEQUENTIAL\_IO.

- w. CE2102P is inapplicable because this implementation supports OPEN with OUT\_FILE mode for SEQUENTIAL\_IO.

- x. CE2102Q is inapplicable because this implementation supports RESET with OUT\_FILE mode for SEQUENTIAL\_IO.

## TEST INFORMATION

- y. CE2102R is inapplicable because this implementation supports OPEN with INOUT\_FILE mode for DIRECT\_IO.
- z. CE2102S is inapplicable because this implementation supports RESET with INOUT\_FILE mode for DIRECT\_IO.
- aa. CE2102T is inapplicable because this implementation supports OPEN with IN\_FILE mode for DIRECT\_IO.
- ab. CE2102U is inapplicable because this implementation supports RESET with IN\_FILE mode for DIRECT\_IO.
- ac. CE2102V is inapplicable because this implementation supports OPEN with OUT\_FILE mode for DIRECT\_IO.
- ad. CE2102W is inapplicable because this implementation supports RESET with OUT\_FILE mode for DIRECT\_IO.
- ae. EE2401D and EE2401G are not applicable because USE\_ERROR is raised when trying to create a file with unconstrained array types.
- af. CE2401H is inapplicable because this implementation does not support CREATE with INOUT\_FILE mode for unconstrained records with default discriminants.
- ag. CE3102E is inapplicable because this implementation supports CREATE with IN\_FILE mode for text files.
- ah. CE3102F is inapplicable because this implementation supports RESET for text files.
- ai. CE3102G is inapplicable because this implementation supports deletion of an external file for text files.
- aj. CE3102I is inapplicable because this implementation supports CREATE with OUT\_FILE mode for text files.
- ak. CE3102J is inapplicable because this implementation supports OPEN with IN\_FILE mode for text files.
- al. CE3102K is inapplicable because this implementation supports OPEN with OUT\_FILE mode for text files.

### 3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting

## TEST INFORMATION

a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 35 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B23004A	B24007A	B24009A	B25002A	B26005A	B27005A
B28003A	B32202A	B32202B	B32202C	B33001A	B36307A
B37004A	B45102A	B49003A	B49005A	B61012A	B62001B
B74304B	B74401F	B74401R	B91004A	B95004A	B95032A
B95069A	B95069B	BA1101B	BC2001D	BC3009A	BC3009C
BD5005B					

The following modifications were made to compensate for legitimate implementation behavior:

- a. At the recommendation of the AVO, the expression "2\*\*T'MANTISSA - 1" on line 262 of test CC1223A was changed to "(2\*\* (T'MANTISSA-1)-1 + 2\*\* (T'MANTISSA-1))" since the previous expression causes an unexpected exception to be raised.

The following tests were graded using a modified evaluation criteria:

- a. BA2001E expects that the non-distinctness of names of subunits with a common ancestor be detected at compile time, but this implementation detects the errors at link time. The AVO ruled that it is also acceptable to make the error detection at link time. Thus, this test is considered to be passed if the intended errors are detected at either compile or link time.
- b. EA3004D and LA3004B both fail to detect an error because pragma INLINE is invoked for a function that is called within a package specification. By re-ordering the files, it may be shown that INLINE indeed has no effect. The AVO has ruled that both are to be run as is, noting that both fail to detect errors where INLINE is invoked from within a package specification. The compilation files are then re-ordered. For EA3004D, the order is 0, 1, 4, 5, 2, 3, 6; and for LA3004B, the order is 0, 1, 5, 6, 2, 3, 4, 7. The tests both execute and produce the expected NOT APPLICABLE result, as though INLINE were not supported at all. Both tests are graded as passed.

### 3.7 ADDITIONAL TESTING INFORMATION

#### 3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the HP 9000 Series 800 Ada Compiler, Version 4.35, was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

#### 3.7.2 Test Method

Testing of the HP 9000 Series 800 Ada Compiler, Version 4.35, using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	HP 9000 Series 800 Model 850
Host operating system:	HP-UX, Version A.B3.10 (release 3.1)
Target computer:	HP 9000 Series 800 Model 850
Target operating system:	HP-UX, Version A.B3.10 (release 3.1)
Compiler:	HP 9000 Series 800 Ada Compiler, Version 4.35

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the HP 9000 Series 800 Model 850. Results were printed from the host computer.

The compiler was tested using command scripts provided by Hewlett Packard Company and reviewed by the validation team. See Appendix E for a complete listing of the compiler options for this implementation. The following list of compiler options includes those options which were invoked by default:



## TEST INFORMATION

-L	Produce an output listing.
-P 66	Set the out page length to 66 lines.
-e 999	Set the maximum number of errors to 999.
+Q	Suppress printing source file name to stdout.

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

### 3.7.3 Test Site

Testing was conducted at Cupertino CA and was completed on 19 October 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

The Hewlett Packard Company has submitted the following  
Declaration of Conformance concerning the HP 9000  
Series 800 Ada Compiler.

## DECLARATION OF CONFORMANCE


Compiler Implementor: Hewlett Packard Company, California Language Lab  
Ada Validation Facility: ASD/SCEL, Wright Patterson AFB, OH 45433-6503  
Ada Compiler Validation Capability (ACVC) Version: 1.10.

### Base Configuration

Base Compiler Name: HP 9000 Series 800 Ada Compiler, Version 4.35.  
Host Architecture ISA: HP 9000 Series 800 Model 850  
Host OS and Version: HP-UX, Version A.B3.10 (release 3.1)  
Target Architecture ISA: HP 9000 Series 800 Model 850  
Target OS and Version: HP-UX, Version Version A.B3.10 (release 3.1)

### Implementer's Declaration


I, the undersigned, representing Hewlett Packard Company, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compilers listed in this declaration. I declare that Hewlett Packard Company is owner of record of the Ada language compilers listed above and, as such, is responsible for maintaining said compilers in conformance to ANSI/MIL-STD-1815A. All certificates and registration for the Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

  
\_\_\_\_\_  
Hewlett Packard Company  
David Graham  
Ada R&D Section Manager

Date: 10/16/89

### Owner's Declaration

I, the undersigned, representing Hewlett Packard Company, take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

  
\_\_\_\_\_  
Hewlett Packard Company  
David Graham  
Ada R&D Section Manager

Date: 10/16/89

## APPENDIX B

### APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the HP 9000 Series 800 Ada Compiler, Version 4.35, as described in this Appendix, are provided by Hewlett Packard Company. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

...

```
type INTEGER is range -2147483648 .. 2147483647;
type SHORT_INTEGER is range -32768 .. 32767;
type SHORT_SHORT_INTEGER is range -128 .. 127;
```

```
type FLOAT is digits 6 range -2#1.111_1111_1111_1111_1111_1111#E+127 ..
2#1.111_1111_1111_1111_1111_1111#E+127;
```

```
type LONG_FLOAT is digits 15 range -2#1.1111_1111_1111_1111_1111_1111_
1111_1111_1111_1111_1111_1111_1111_1111#E+1023 .. 2#1.1111_1111_1111_1111_
1111_1111_1111_1111_1111_1111_1111_1111#E+1023;
```

```
type DURATION is delta 2#0.000_000_000_000_01# range -86_400.0 .. 86_400.0;
```

...

end STANDARD;

# Appendix F

## Implementation-Dependent Characteristics for the HP 9000 Series 800 Development System

The Ada programming language is described in the *Reference Manual for the Ada Programming Language* (hereafter called *Ada RM*). This manual, *Reference Manual for the Ada Programming Language, Appendix F*, summarizes the implementation dependencies of the HP Ada Development System on the HP 9000 Series 800 Computer System.

This manual describes the following:

- HP implementation-dependent pragmas and attributes.
- Specifications of the packages SYSTEM and STANDARD.
- Instructions on using type representation clauses to fully specify the layout of data objects in memory.
- Restrictions on unchecked type conversions.
- Implementation-dependent characteristics of input/output packages.
- Information about HP-UX signals and the Ada runtime environment.
- Instruction and examples on calling external subprogram written in HP-PA Assembly Language, HP C, HP FORTRAN 77, and HP Pascal.

## F 1. Implementation Supported Pragmas

This section describes the predefined language pragmas and the Ada/800 implementation-specific pragmas. Table F-1 lists these pragmas.

**Table F-1. Ada/800 Pragmas**

Action	Pragma Name
Interface with subprograms written in other languages	INTERFACE INTERFACE_NAME
Support text processing tools	INDENT LIST PAGE
Determine the layout of array and record types in memory	PACK IMPROVE
Direct the compiler to generate different code than what is normally generated	ELABORATE INLINE SUPPRESS
Affect tasking programs	PRIORITY SHARED

Section F 1.6 lists predefined pragmas not implemented in Ada/800.

## F 1.1 Interfacing the Ada Language with Other Languages

Your Ada programs can call subprograms written in other languages when you use the predefined pragmas `INTERFACE` and `INTERFACE_NAME`. Ada/800 supports subprograms written in these languages:

- HP-PA Assembly Language
- HP C
- HP Pascal
- HP FORTRAN 77 for HP 9000 Series 800 computers

Compiler products from vendors other than Hewlett-Packard may not conform to the parameter passing conventions given below. See Section F 11 for detailed information, instructions, and examples for interfacing your Ada programs with the above languages.

### F1.1.1 Pragma `INTERFACE`

The pragma `INTERFACE` (*Ada RM*, Section 13.9) informs the compiler that a non-Ada external subprogram will be supplied when the Ada program is linked. Pragma `INTERFACE` specifies the programming language used in the external subprogram and the name of the Ada interfaced subprogram. The corresponding parameter calling convention to be used in the interface is implicitly defined in the language specification.

#### Syntax

```
pragma INTERFACE ( Language_name, Ada_subprogram_name );
```

where:

*Language\_name* is one of `ASSEMBLER`, `C`, `PASCAL`, or `FORTRAN`.

*Ada\_subprogram\_name* is the name used within the Ada program when referring to the interfaced external subprogram.

It is not possible to supply a pragma `INTERFACE` to a library-level subprogram. Any subprogram that a pragma `INTERFACE` applies to must be contained within an Ada compilation unit, usually a package.

### F 1.1.2 Pragma `INTERFACE_NAME`

Ada/800 provides the implementation-defined pragma `INTERFACE_NAME` to associate an alternative name with a non-Ada external subprogram that has been specified to the Ada program by the pragma `INTERFACE`.

#### Syntax

```
pragma INTERFACE_NAME ( Ada_subprogram_name, "External_subprogram_name" );
```

where:

*Ada\_subprogram\_name* is the name used within the Ada program when referring to the interfaced external subprogram.

*External\_subprogram\_name* is the external name used outside the Ada program.

You must use pragma `INTERFACE_NAME` whenever the interfaced subprogram name contains characters not acceptable within Ada identifiers or when the interfaced subprogram name contains uppercase letter(s). You can also use a pragma `INTERFACE_NAME` if you want your Ada subprogram name to be different than the external subprogram name.

If a pragma `INTERFACE_NAME` is not supplied, the external subprogram name is the name of the Ada subprogram specified in the pragma `INTERFACE`, with all alphabetic characters shifted to lowercase letters.

Pragma `INTERFACE_NAME` is allowed at the same places in an Ada program as pragma `INTERFACE` ( see *Ada RM*, Section 13.9.) Pragma `INTERFACE_NAME` must follow the declaration of the corresponding pragma `INTERFACE` and must be within the same declarative part, although it need *not* immediately follow that declaration.



### F 1.1.3 Example of INTERFACE and INTERFACE\_NAME

The following example illustrates the INTERFACE and INTERFACE\_NAME pragmas.

```

package SAMPLE_LIB is

    function SAMPLE_DEVICE (X : INTEGER) return INTEGER;
    function PROCESS_SAMPLE (X : INTEGER) return INTEGER;

private

    pragma INTERFACE (ASSEMBLER, SAMPLE_DEVICE );
    pragma INTERFACE (C, PROCESS_SAMPLE);

    pragma INTERFACE_NAME (SAMPLE_DEVICE, "Dev10" );
    pragma INTERFACE_NAME (PROCESS_SAMPLE, "DoSample" );

end SAMPLE_LIB;
```

This example defines two Ada subprograms that are known in Ada code as SAMPLE\_DEVICE and PROCESS\_SAMPLE. When a call to SAMPLE\_DEVICE is executed, the program will generate a call to the externally supplied assembly function Dev10. Likewise, when a call to PROCESS\_SAMPLE is executed, the program will generate a call to the externally supplied HP C function DoSample.

By using the pragma INTERFACE\_NAME, the names for the external subprograms to associate with the Ada subprogram are explicitly identified. If pragma INTERFACE\_NAME had not been used, the two external names referenced would be sample\_device and process\_sample.

### F 1.1.4 Additional Information on INTERFACE and INTERFACE\_NAME

Either an object file (for binding and linking with the same command) or an object library (for binding and linking separately) that defines the external subprograms must be provided as a command line parameter to the Ada binder. The command line parameter must be provided to the linker *ld(1)* if you call the linker separately. If you do not provide an object file that contains the definition for the external subprogram, the HP-UX linker, *ld(1)*, will issue an error message.

To avoid conflicts with the Ada runtime system, the names of interfaced external routines should not begin with the letters "alsy", "\_Ada", or "RTS" because the Ada runtime system prefixes its internal routines with these prefixes.

## Implementation-Dependent Characteristics

When you want to call an HP-UX system call from Ada code, you should use a pragma `INTERFACE` with `C` as the language name. You might need to use a pragma `INTERFACE_NAME` to explicitly supply the external name. This external name must be the same as the name of the system call that you want to call. ( See Section 2 of the *HP-UX Reference* for details.) In this case it is not necessary to provide the C object file to the binder, because it will be found automatically when the linker searches the system library.

When you want to call an HP-UX library function from Ada code, you should use a pragma `INTERFACE` with `C` as the language name. You should use pragma `INTERFACE_NAME` to explicitly supply the external name. This external name must be exactly the same as the name of the library function. ( See Section 3 of the *HP-UX Reference* for details. ) If your library function is located in either the Standard C Library or the Math Library, it is not necessary to provide the object library to the binder because the binder always requests that the linker search these two libraries. If your library function is located in any of the other standard libraries, you must provide the appropriate `-lx` option to the binder that the binder will pass onto the linker as a request to search the specified library.

See Section F 11 for additional information on using pragma `INTERFACE` and pragma `INTERFACE_NAME`.

## F 1.2 Using Text Processing Tools

The pragma `INDENT` is a formatting command that affects the HP supplied reformatter, *ada.format(1)*. This pragma does not affect the compilation listing output of the compiler. The pragmas `LIST` and `PAGE` are formatting commands that affect the compilation listing output of the compiler.

### F 1.2.1 Pragma `INDENT`

Ada/800 provides the implementation-defined pragma `INDENT` to assist in reformatting Ada source code. You can place these pragmas in the source code to control the actions of *ada.format(1)*.

#### Syntax

```
pragma INDENT ( ON | OFF );
```

#### Parameters

Parameter	Description
OFF	<i>ada.format</i> does not modify the source lines after the pragma.
ON	<i>ada.format</i> resumes its action after the pragma.

The default for pragma `INDENT` is `ON`.

### F 1.2.2 Pragma LIST

The pragma LIST affects only the compilation listing output of the compiler. It specifies that the listing of the compilation is to be continued or suspended until a LIST pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing. The compilation listing feature of the compiler is enabled by issuing one of the compiler options -L or -B to the *ada(1)* command.

#### Syntax

```
pragma LIST ( ON | OFF );
```

#### Parameters

Parameter	Description
OFF	The listing of the compilation is suspended after the pragma.
ON	The listing of the compilation is resumed and the pragma is listed.

The default for pragma LIST is ON.

### F 1.2.3 Pragma PAGE

The pragma PAGE affects the compilation listing output of the compiler. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).

#### Syntax

```
pragma PAGE;
```

### F 1.3 Affecting the Layout of Array and Record Types

The pragmas PACK and IMPROVE affect the layout of array and record types in memory.

#### F 1.3.1 Pragma PACK

The pragma PACK takes the simple name of an array type as its only argument. The allowed positions for this pragma and the restrictions on the named type are governed by the same rules as for a representation clause. The pragma specifies that storage minimization should be the main criterion when selecting the representation of the given type.

##### Syntax

```
pragma PACK ( array_type_name );
```

The pragma PACK is not implemented for record types on Ada/800. You can use a record representation clause to minimize the storage requirements for a record type.

The pragma PACK is discussed further in Section F 4.7, "Array Types."

#### F 1.3.2 Pragma IMPROVE

The pragma IMPROVE, an *implementation-defined pragma*, suppresses implicit components in a record type.

##### Syntax

```
pragma IMPROVE ( TIME | SPACE , [ON =>] record_type_name );
```

The default for pragma IMPROVE is TIME. This pragma is discussed further in Section F 4.8, "Record Types."

### F 1.4 Generating Code

The pragmas ELABORATE, INLINE, and SUPPRESS direct the compiler to generate different code than would have been normally generated. These pragmas can change the run time behavior of an Ada program unit.

### F 1.4.1 Pragma ELABORATE

The pragma ELABORATE is used when a dependancy upon elaboration order exists. Normally the Ada compiler is given the freedom to elaborate library units in any order. This pragma specifies that the bodies for each of the library units named in the pragma must be elaborated before the current compilation unit. If the current compilation unit is a subunit, the bodies of the named library units must be elaborated before the body of the parent of the current subunit.

#### Syntax

```
pragma ELABORATE ( library_unit_name
                  [, library_unit_name ] ... );
```

This pragma takes as its arguments one or more simple names, each of which denotes a library unit. This pragma is only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument must be the simple name of a library unit that was identified by the context clause. (See the *Ada RM*, Section 10.5, for additional information on elaboration of library units.)

### F 1.4.2 Pragma INLINE

The pragma INLINE specifies that the subprogram bodies should be expanded inline at each call whenever possible; in the case of a generic subprogram, the pragma applies to calls of its instantiations. If the subprogram name is overloaded, the pragma applies to every overloaded subprogram. Note that pragma INLINE has no effect on function calls appearing inside package specifications.

#### Syntax

```
pragma INLINE ( subprogram_name [, subprogram_name ] ... );
```

This pragma takes as its arguments one or more names, each of which is either the name of a subprogram or the name of a generic subprogram. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. See the *Ada RM*, Section 6.3.2, for additional information on inline expansion of subprograms.

This pragma can be suppressed at compile time by issuing the compiler option `-I` to the *ada(1)* command.

### F 1.4.3 Pragma SUPPRESS

The pragma SUPPRESS allows the compiler to omit the given check from the place of the pragma to the end of the declarative region associated with the innermost enclosing block statement or program unit. For a pragma given in a package specification, the permission extends to the end of the scope of the named entity.

#### Syntax

```
pragma SUPPRESS ( check_identifier [, [ON =>] name] );
```

The pragma SUPPRESS takes as arguments the identifier of a check and optionally the name of either an object, a type or subtype, a subprogram, a task unit, or a generic unit. This pragma is only allowed at the place of a declarative item in a declarative part or a package specification.

If the pragma includes a name, the permission to omit the given check is further restricted: it is given only for operations on the named object or on all objects of the base type of a named type or subtype; for calls of a named subprogram; for activations of tasks of the named task type; or for instantiations of the given generic unit. ( See the *Ada RM*, Section 11.7, for additional information on suppressing run time checks. )

The compiler can be directed to suppress all run time checks by issuing the compiler option -R to the *ada(1)* command. The compiler can also be directed to suppress all run time checks except for stack checks by issuing the compiler option -C to the *ada(1)* command.

### F 1.5 Affecting Run Time Behavior

The pragmas PRIORITY and SHARED affect the run time behavior of a tasking program.

#### F 1.5.1 Pragma PRIORITY

The pragma PRIORITY specifies the priority to be used for the task or tasks of the task type. When the pragma is applied within the outermost declarative part of the main subprogram, it specifies the priority to be used for the environment task, which is the task that encloses the main subprogram. If a pragma PRIORITY is applied to a subprogram that is not the main subprogram, it is ignored.

#### Syntax

```
pragma PRIORITY ( static_expression );
```

The pragma PRIORITY takes as its argument a static expression of the predefined integer subtype PRIORITY. For Ada/800, the range of the subtype PRIORITY is 1 to 127. This pragma is only allowed within the specification of a task unit or within the outermost declarative part of the main subprogram.

These task priorities are only relative to other Ada tasks that are concurrently executing with the environment task. This pragma does *not* change the priority of an Ada task or the Ada environment task relative to other HP-UX processes. All the Ada tasks execute within a single HP-UX process. This HP-UX process executes together with other HP-UX processes and is scheduled by the HP-UX kernel. To change the priority of an HP-UX process, see the command *nice(1)*. See the *Ada RM*, Section 9.8, for additional information on task priorities.

### F 1.5.2 Pragma SHARED

The pragma SHARED specifies that every read or update of the variable is a synchronization point for that variable. The type for the variable object is limited to scalar or access types because each read or update must be implemented as an indivisible operation.

The effect of pragma SHARED on a variable object is to suppress the promotion of this object to a register by the compiler. The compiler suppresses this optimization and ensures that any reference to the variable always retrieves the value stored by the most recent update operation.

#### Syntax

```
pragma SHARED ( variable_simple_name );
```

The pragma SHARED takes as its argument a simple name of a variable. This pragma is only allowed for a variable declared by an object declaration and whose type is a scalar or access type; the variable declaration and the pragma must both occur (in this order) within the same declarative part or package specification.

See the *Ada RM*, Section 9.11, for additional information on shared variables.

### F 1.6 Pragmas Not Implemented

The following predefined language pragmas are not implemented and will issue a warning at compile time:

```
pragma CONTROLLED ( access_type_simple_name );
```

```
pragma MEMORY_SIZE ( numeric_literal );
```

```
pragma OPTIMIZE ( TIME | SPACE );
```

```
pragma STORAGE_UNIT ( numeric_literal );
```

```
pragma SYSTEM_NAME ( enumeration_literal );
```

See the *Ada RM*, Appendix B, for additional information on these predefined language pragmas.

## F 2. Implementation-Dependent Attributes

In addition to the representation attributes discussed in the *Ada RM*, Section 13.7.2, there are five implementation-defined representation attributes:

```
'OFFSET  
'RECORD_SIZE  
'VARIANT_INDEX  
'ARRAY_DESCRIPTOR  
'RECORD_DESCRIPTOR
```

These implementation-defined attributes are only used to refer to implicit components of record types inside a record representation clause. Using these attributes outside of a record representation clause will cause a compiler error message. For additional information, see Section F 4.8, "Record Types".

### F 2.1 Limitation of the Attribute 'ADDRESS

The attribute 'ADDRESS is implemented for all entities that have meaningful addresses. The compiler will issue the following warning message when the prefix for the attribute 'ADDRESS refers to an object that has a meaningless address:

```
The prefix of the 'ADDRESS attribute denotes a program unit that  
has no meaningful address: the result of such an evaluation is  
SYSTEM.NULL_ADDRESS.
```

The following entities do not have meaningful addresses and will cause the above compilation warning if used as a prefix to 'ADDRESS:

- A constant that is implemented as an immediate value (that is, a constant that does not have any space allocated for it).
- A package identifier that is not a library unit or a subunit.
- A function that renames an enumeration literal.

Additionally, the attribute 'ADDRESS when applied to a task type will return different values, depending upon the elaboration time of the task body. In particular, the value returned by the attribute 'ADDRESS before the task body is elaborated is zero. After elaboration of the task body, the address of the task body will be returned. It is recommended that 'ADDRESS not be applied to a task until after the task body has been elaborated.



### F 3. The SYSTEM and STANDARD Packages

This section contains a complete listing of the two predefined library packages: SYSTEM and STANDARD. These packages both contain implementation-dependent specifications.

#### F 3.1 The Package SYSTEM

The specification of the predefined library package SYSTEM follows:

```
-- Standard Ada definitions

type NAME is ( HP9000_800 );
SYSTEM_NAME: constant NAME := HP9000_800;

STORAGE_UNIT: constant := 8;

MEMORY_SIZE: constant := (2**31)-1;

MIN_INT:      constant := -(2**31);

MAX_INT:      constant := 2**31-1;

MAX_DIGITS:   constant := 15;

MAX_MANTISSA: constant := 31;

FINE_DELTA:   constant := 2#1.0#e-31;

TICK: constant := 0.010;      -- 10 milliseconds

subtype PRIORITY is INTEGER range 1..127 ;

type ADDRESS is private;

NULL_ADDRESS : constant ADDRESS; --set to NULL

-- exception to be raised when
-- (1) the space id of an address changes after addition or subtraction
-- (2) the space id's of two addresses are not the same during comparison
ADDRESS_ERROR: exception;

-- Address arithmetic

function TO_INTEGER (LEFT : ADDRESS) return INTEGER;

function TO_ADDRESS (LEFT : INTEGER) return ADDRESS;

-- Note that ADDRESS + ADDRESS is not supported
function "+" (LEFT : INTEGER; RIGHT : ADDRESS) return ADDRESS;
function "+" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;
```

## Implementation-Dependent Characteristics

```
-- Note that INTEGER - ADDRESS is not supported
function "-" (LEFT : ADDRESS; RIGHT : ADDRESS) return INTEGER;
function "-" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;

function "<" (LEFT : ADDRESS; RIGHT : ADDRESS) return BOOLEAN;
function "<=" (LEFT : ADDRESS; RIGHT : ADDRESS) return BOOLEAN;
function ">" (LEFT : ADDRESS; RIGHT : ADDRESS) return BOOLEAN;
function ">=" (LEFT : ADDRESS; RIGHT : ADDRESS) return BOOLEAN;

function "mod" (LEFT : ADDRESS; RIGHT : POSITIVE) return NATURAL;

function IS_NULL (LEFT : ADDRESS) return BOOLEAN;

function ALIGN (LEFT: ADDRESS) return ADDRESS;
-- align the given address up to four bytes boundary

function ALIGN (LEFT: ADDRESS; ALIGNMENT: INTEGER) return ADDRESS;
-- align the given address up or down to the alignment boundary

function IS_ALIGNED (LEFT: ADDRESS; ALIGNMENT: POSITIVE)
return BOOLEAN;

procedure COPY (FROM : ADDRESS; TO : ADDRESS; SIZE : NATURAL);
-- Copy SIZE storage units.

-- Functions to provide READ/WRITE operations in memory.

generic
    type ELEMENT_TYPE is private;
function FETCH (FROM : ADDRESS) return ELEMENT_TYPE;
-- Return the bit pattern stored at address FROM, as a value of
-- the given type.
-- WARNING: The use of this function with unconstrained array types
-- can result in erroneous program execution. The target object of
-- this function may be damaged or unallocated memory may be accessed
-- resulting in an exception being raised or the program being
-- aborted.

generic
    type ELEMENT_TYPE is private;
procedure STORE (INTO : ADDRESS; OBJECT : ELEMENT_TYPE);
-- Store the bit pattern representing the value of OBJECT, at
-- address INTO.
-- WARNING: The use of this procedure with unconstrained array types
-- can result in erroneous program execution. The target object of
-- this procedure may be damaged or unallocated memory may be
-- accessed resulting in an exception begin raised or the program
-- being aborted.
```

**private**

-- private part of package SYSTEM

...

**end SYSTEM;**

## F 3.2 The Package STANDARD

The specification of the predefined library package STANDARD follows:

**package** STANDARD is

```
-- The operators that are predefined for the types declared in this
-- package are given in comments since they are implicitly declared.
-- Italics are used for pseudo-names of anonymous types (such as
-- universal_real, universal_integer, and universal_fixed)
-- and for undefined information (such as any_fixed_point_type).

-- Predefined type BOOLEAN
type BOOLEAN is (FALSE, TRUE);

-- The predefined relational operators for this type are as follows
-- (these are implicitly declared):
-- function "=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "<" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function ">" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

-- The predefined logical operands and the predefined logical
-- negation operator are as follows (these are implicitly
-- declared):

-- function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "or" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

-- function "not" (RIGHT : BOOLEAN) return BOOLEAN;

-- Predefined universal types

-- type universal_integer is predefined;

-- The predefined operators for the type universal_integer are as
-- follows (these are implicitly declared):
-- function "=" (LEFT, RIGHT : universal_integer) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : universal_integer) return BOOLEAN;
-- function "<" (LEFT, RIGHT : universal_integer) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : universal_integer) return BOOLEAN;
-- function ">" (LEFT, RIGHT : universal_integer) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : universal_integer) return BOOLEAN;

-- function "+" (RIGHT : universal_integer) return universal_integer;
-- function "-" (RIGHT : universal_integer) return universal_integer;
-- function "abs" (RIGHT : universal_integer) return universal_integer;
```

```

-- function "+" (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "-" (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "*" (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "/" (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "rem" (LEFT, RIGHT : universal_integer) return universal_integer;
-- function "mod" (LEFT, RIGHT : universal_integer) return universal_integer;

-- function "**" (LEFT : universal_integer;
--              RIGHT : INTEGER) return universal_integer;

-- type universal_real is predefined;

-- The predefined operators for the type universal_real are as follows
-- (these are implicitly declared):
-- function "=" (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function "<" (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function ">" (LEFT, RIGHT : universal_real) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : universal_real) return BOOLEAN;

-- function "+" (RIGHT : universal_real) return universal_real;
-- function "-" (RIGHT : universal_real) return universal_real;
-- function "abs" (RIGHT : universal_real) return universal_real;

-- function "+" (LEFT, RIGHT : universal_real) return universal_real;
-- function "-" (LEFT, RIGHT : universal_real) return universal_real;
-- function "*" (LEFT, RIGHT : universal_real) return universal_real;
-- function "/" (LEFT, RIGHT : universal_real) return universal_real;

-- function "**" (LEFT : universal_real;
--              RIGHT : INTEGER) return universal_real;

-- In addition, the following operators are predefined for universal types:

-- function "*" (LEFT : universal_integer;
--              RIGHT : universal_real) return universal_real;
-- function "*" (LEFT : universal_real;
--              RIGHT : universal_integer) return universal_real;
-- function "/" (LEFT : universal_real;
--              RIGHT : universal_integer) return universal_real;

-- type universal_fixed is predefined;

-- The only operators declared for this type are:
-- function "*" (LEFT : any_fixed_point_type;
--              RIGHT : any_fixed_point_type) return universal_fixed;
-- function "/" (LEFT : any_fixed_point_type;
--              RIGHT : any_fixed_point_type) return universal_fixed;

-- Predefined and additional integer types

type SHORT_SHORT_INTEGER is range -128 .. 127; -- 8 bits long
-- this is equivalent to -(2**7) .. (2**7)-1

```

## Implementation-Dependent Characteristics

```
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "=" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function "<" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function ">" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return BOOLEAN;

-- function "+" (RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;
-- function "-" (RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;
-- function "abs" (RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;

-- function "+" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;
-- function "-" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;
-- function "*" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;
-- function "/" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;
-- function "rem" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;
-- function "mod" (LEFT, RIGHT : SHORT_SHORT_INTEGER) return SHORT_SHORT_INTEGER;

-- function "**" (LEFT : SHORT_SHORT_INTEGER;
--              RIGHT : INTEGER) return SHORT_SHORT_INTEGER;
```

type SHORT\_INTEGER is range -32\_768 .. 32\_767; --16 bits long

```
-- this is equivalent to -(2**15) .. (2**15)-1
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function "<" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function ">" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : SHORT_INTEGER) return BOOLEAN;

-- function "+" (RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "-" (RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "abs" (RIGHT : SHORT_INTEGER) return SHORT_INTEGER;

-- function "+" (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "-" (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "*" (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "/" (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "rem" (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;
-- function "mod" (LEFT, RIGHT : SHORT_INTEGER) return SHORT_INTEGER;

-- function "**" (LEFT : SHORT_INTEGER;
--              RIGHT : INTEGER) return SHORT_INTEGER;
```

type INTEGER is range -2\_147\_483\_648 .. 2\_147\_483\_647;

-- type INTEGER is 32 bits long

```

-- this is equivalent to  $-(2^{31}) \dots (2^{31})-1$ 
-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : INTEGER) return BOOLEAN;

-- function "+" (RIGHT : INTEGER) return INTEGER;
-- function "-" (RIGHT : INTEGER) return INTEGER;
-- function "abs" (RIGHT : INTEGER) return INTEGER;

-- function "+" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "-" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "*" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "/" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;

-- function "**" (LEFT : INTEGER; RIGHT : INTEGER) return INTEGER;

-- Predefined INTEGER subtypes
subtype NATURAL is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;

-- Predefined and additional floating point types

type FLOAT is digits 6 range -- 32 bits long
-2#1.111_1111_1111_1111_1111_1111#E+127 ..
 2#1.111_1111_1111_1111_1111_1111#E+127;
--
-- This is equivalent to  $-(2.0 - 2.0^{(-23)}) * 2.0^{127} \dots$ 
--  $+(2.0 - 2.0^{(-23)}) * 2.0^{127}$ 
--
-- This is approximately equal to the decimal range:
-- -3.402823E+38 .. +3.402823E+38

-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : FLOAT) return BOOLEAN;

-- function "+" (RIGHT : FLOAT) return FLOAT;
-- function "-" (RIGHT : FLOAT) return FLOAT;
-- function "abs" (RIGHT : FLOAT) return FLOAT;

```

## Implementation-Dependent Characteristics

```

-- function "+" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "-" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "*" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "/" (LEFT, RIGHT : FLOAT) return FLOAT;

-- function ":" (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;

type LONG_FLOAT is digits 15 range -- 64 bits long
-2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E+1023
..
2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E+1023;
--
-- This is equivalent to -(2.0 - 2.0**(-52)) * 2.0**1023 ..
--      +(2.0 - 2.0**(-52)) * 2.0**1023 ..
-- This is approximately equal to the decimal range:
-- -1.797693134862315E+308 .. +1.797693134862315E+308

-- The predefined operators for this type are as follows
-- (these are implicitly declared):
-- function "=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "<" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function ">" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;

-- function "+" (RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "-" (RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "abs" (RIGHT : LONG_FLOAT) return LONG_FLOAT;

-- function "+" (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "-" (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "*" (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "/" (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;

-- function "**" (LEFT : LONG_FLOAT; RIGHT : INTEGER) return LONG_FLOAT;

--This implementation does not provide any other floating point types

-- Predefined type DURATION
type DURATION is delta 2#0.000_000_000_000_01# range -86_400.0 .. 86_400.0;
--
-- DURATION'SMALL derived from this delta is 2.0**(-14) , which is the
-- maximum precision that an object of type DURATION can have and still
-- be representable in this implementation. This has an approximate
-- decimal equivalent of 0.000061 (61 microseconds).
-- The predefined operators for the type DURATION are the same as for any
-- fixed point type.

-- This implementation provides many anonymous predefined fixed point
-- types. They consist of fixed point types whose "small" value is
-- a power of 2.0 and whose mantissa can be expressed using 31 or less
-- binary digits.

```



-- Predefined type CHARACTER

-- The following lists characters for the standard ASCII character set.  
 -- Character literals corresponding to control characters are not  
 -- identifiers; they are indicated in italics in this section.

type CHARACTER is

```
( nul, soh, stx, etx, eot, enq, ack, bel,
  bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
  dle, dc1, dc2, dc3, dc4, nak, syn, etb,
  can, em,  sub, esc, fs,  gs,  rs,  us,

  ' ', '!', '"', '#', '$', '%', '&', "'",
  '(', ')', '*', '+', ',', '-', '.', '/',
  '0', '1', '2', '3', '4', '5', '6', '7',
  '8', '9', ':', ';', '<', '=', '>', '?',

  '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
  'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
  'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
  'X', 'Y', 'Z', '[', '\', ']', '^', '_',

  'a', 'b', 'c', 'd', 'e', 'f', 'g',
  'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
  'x', 'y', 'z', '{', '|', '}', '~', del);
```

--The predefined operators for the type CHARACTER are the same as  
 --for any enumeration type.

-- Predefined type STRING (RM 3.6.3)

type STRING is array (POSITIVE range <>) of CHARACTER;

-- The predefined operators for this type are as follows:

```
-- function "=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : STRING) return BOOLEAN;
```

-- Predefined catenation operators

```
-- function "&" (LEFT : STRING; RIGHT : STRING) return STRING;
-- function "&" (LEFT : CHARACTER; RIGHT : STRING) return STRING;
-- function "&" (LEFT : STRING; RIGHT : CHARACTER) return STRING;
-- function "&" (LEFT : CHARACTER; RIGHT : CHARACTER) return STRING;
```

-- Predefined exceptions

```
CONSTRAINT_ERROR : exception;
NUMERIC_ERROR    : exception;
PROGRAM_ERROR    : exception;
STORAGE_ERROR    : exception;
TASKING_ERROR    : exception;
```

## Implementation-Dependent Characteristics

-- Predefined package ASCII  
package ASCII is

-- Control characters

NUL : constant CHARACTER := *nul*;  
SOH : constant CHARACTER := *soh*;  
STX : constant CHARACTER := *stx*;  
ETX : constant CHARACTER := *etx*;  
EOT : constant CHARACTER := *eot*;  
ENQ : constant CHARACTER := *enq*;  
ACK : constant CHARACTER := *ack*;  
BEL : constant CHARACTER := *bel*;  
BS : constant CHARACTER := *bs*;  
HT : constant CHARACTER := *ht*;  
LF : constant CHARACTER := *lf*;  
VT : constant CHARACTER := *vt*;  
FF : constant CHARACTER := *ff*;  
CR : constant CHARACTER := *cr*;  
SO : constant CHARACTER := *so*;  
SI : constant CHARACTER := *si*;  
DLE : constant CHARACTER := *dle*;  
DC1 : constant CHARACTER := *dc1*;  
DC2 : constant CHARACTER := *dc2*;  
DC3 : constant CHARACTER := *dc3*;  
DC4 : constant CHARACTER := *dc4*;  
NAK : constant CHARACTER := *nak*;  
SYN : constant CHARACTER := *syn*;  
ETB : constant CHARACTER := *etb*;  
CAN : constant CHARACTER := *can*;  
EM : constant CHARACTER := *em*;  
SUB : constant CHARACTER := *sub*;  
ESC : constant CHARACTER := *esc*;  
FS : constant CHARACTER := *fs*;  
GS : constant CHARACTER := *gs*;  
RS : constant CHARACTER := *rs*;  
US : constant CHARACTER := *us*;  
DEL : constant CHARACTER := *del*;

-- other characters

EXCLAM : constant CHARACTER := '!';  
QUOTATION : constant CHARACTER := '"';  
SHARP : constant CHARACTER := '#';  
DOLLAR : constant CHARACTER := '\$';  
PERCENT : constant CHARACTER := '%';  
AMPERSAND : constant CHARACTER := '&';  
COLON : constant CHARACTER := ':';  
SEMICOLON : constant CHARACTER := ';';  
QUERY : constant CHARACTER := '?';  
AT\_SIGN : constant CHARACTER := '@';  
L\_BRACKET : constant CHARACTER := '[';  
BACK\_SLASH : constant CHARACTER := '\';  
R\_BRACKET : constant CHARACTER := ']';  
CIRCUMFLEX : constant CHARACTER := '^';  
UNDERLINE : constant CHARACTER := '\_';

```

GRAVE      : constant CHARACTER := '`';
L_BRACE    : constant CHARACTER := '{';
BAR        : constant CHARACTER := '|';
R_BRACE    : constant CHARACTER := '}';
TILDE     : constant CHARACTER := '~';

```

```
-- Lower case letters
```

```

LC_A : constant CHARACTER := 'a';
LC_B : constant CHARACTER := 'b';
LC_C : constant CHARACTER := 'c';
LC_D : constant CHARACTER := 'd';
LC_E : constant CHARACTER := 'e';
LC_F : constant CHARACTER := 'f';
LC_G : constant CHARACTER := 'g';
LC_H : constant CHARACTER := 'h';
LC_I : constant CHARACTER := 'i';
LC_J : constant CHARACTER := 'j';
LC_K : constant CHARACTER := 'k';
LC_L : constant CHARACTER := 'l';
LC_M : constant CHARACTER := 'm';
LC_N : constant CHARACTER := 'n';
LC_O : constant CHARACTER := 'o';
LC_P : constant CHARACTER := 'p';
LC_Q : constant CHARACTER := 'q';
LC_R : constant CHARACTER := 'r';
LC_S : constant CHARACTER := 's';
LC_T : constant CHARACTER := 't';
LC_U : constant CHARACTER := 'u';
LC_V : constant CHARACTER := 'v';
LC_W : constant CHARACTER := 'w';
LC_X : constant CHARACTER := 'x';
LC_Y : constant CHARACTER := 'y';
LC_Z : constant CHARACTER := 'z';

```

```
end ASCII;
```

```
end STANDARD;
```

## F 4. Type Representation

This section explains how data objects are represented and allocated by the HP Ada compiler for the HP 9000 Series 800 Computer System and how to control this using representation clauses.

The representation of a data object is closely connected with its type. Therefore, this section successively describes the representation of enumeration, integer, floating point, fixed point, access, task, array and record types. For each class of type, the representation of the corresponding data object is described. Except for array and record types, the description for each class of type is independent of the others. Because array and record types are composite types, it is necessary to understand the representation of their components.

Ada/800 provides several methods to control the layout and size of data objects; these methods are listed in Table F-2.

Table F-2. Methods to Control Layout and Size of Data Objects

Method	Type Used On
pragma PACK	array type
pragma IMPROVE	record type
enumeration representation clause	enumeration type
record representation clause	record type
size specification clause	any type

### F 4.1 Enumeration Types

Syntax (Enumeration representation clause)

*for enumeration-type-name use aggregate;*

The aggregate used to specify this mapping is written as a one-dimensional aggregate, for which the index subtype is the enumeration type and the component type is *universal\_integer*. An *others* choice is *not* permitted in this aggregate.

### Internal Codes of Enumeration Literals

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the enumeration literal. Thus, for an enumeration type with  $n$  elements, the internal codes are the integers 0, 1, 2, ...,  $n-1$ .

An enumeration representation clause can be provided to specify the value of each internal code as described in the *Ada RM*, Section 13.3. The values used to specify the internal codes must be in the range  $-(2^{31})$  to  $(2^{31})-1$ .

The following example illustrates the use of an enumeration representation clause.

### Example

```
type COLOR is (RED, ORANGE, YELLOW, GREEN, AQUA, BLUE, VIOLET);

for COLOR use
    (RED      => 10,
     ORANGE   => 20,
     YELLOW   => 40,
     GREEN    => 80,
     AQUA     => 160,
     BLUE     => 320,
     VIOLET   => 640);
```

In the above example, the internal representation for GREEN will be the integer 80. The attributes 'SUCC and 'PRED will still return YELLOW and AQUA, respectively. Also, the *Ada RM*, in section 13.3(6), states that the 'POS attribute will still return the positional value of the enumeration literal. In the case of GREEN, the value that 'POS returns will be 3 and not 80. The only way to examine the internal representation of the enumeration literal is to write the value to a file or use UNCHECKED\_CONVERSION to examine the value in memory.

## Implementation-Dependent Characteristics

### Minimum Size of an Enumeration Type or Subtype

The minimum size of an enumeration subtype is the minimum number of bits that is necessary for representing the internal codes in normal binary form.

A static subtype of a null range has a minimum size of one. Otherwise, define  $m$  and  $M$  to be the smallest and largest values for the internal codes values of the subtype. The minimum size  $L$  is determined as follows:

Value of $m$	Calculation of $L$ - smallest positive integer such that:	Representation
$m \geq 0$	$M \leq (2^{**}L) - 1$	Unsigned
$m < 0$	$-(2^{**}(L-1)) \leq m$ and $M \leq (2^{**}(L-1))-1$	Signed two's complement

### Example

```
type COLOR is (RED, ORANGE, YELLOW, GREEN, AQUA, BLUE, VIOLET);
-- The minimum size of COLOR is 3 bits.
```

```
subtype SUNNY_COLOR is COLOR range ORANGE .. YELLOW;
-- The minimum size of COLOR is 2 bits.
-- because the internal code for YELLOW is 2
-- and  $(2^{**1})-1 \leq 2 \leq (2^{**2})-1$ 
```

```
type TEMPERATURE is (FREEZING, COLD, MILD, WARM, HOT);
for TEMPERATURE use
```

```
    (FREEZING => -10,
     COLD => 0,
     MILD => 10,
     WARM => 20,
     HOT => 30);
```

```
-- The minimum size of TEMPERATURE is 6 bits
-- because with six bits we can represent signed
-- integers between -32 and 31.
```

### Size of an Enumeration Type

When no size specification is applied to an enumeration type, the objects of that type are represented as signed machine integers. The HP 9000 Series 800 Computer System provides 8-bit, 16-bit and 32-bit integers, and the compiler automatically selects the smallest signed machine integer which can hold all of the internal codes of the enumeration type. Thus, the default size for enumeration types with 128 or less elements is 8 bits, the default size for enumeration types with 129 to 32768 elements is 16 bits. Because this implementation does not support enumeration types with more than 32768 elements, a size specification or enumeration representation clause must be used for enumeration types that use a 32-bit representation.

When a size specification is applied to an enumeration type, this enumeration type and all of its subtypes have the size specified by the length clause. The size specification must specify a value greater than or equal to the minimum size of the type. Note that if the size specification specifies the minimum size and none of the internal codes are negative integers, the internal representation will be that of an unsigned type. Thus, when using a size specification of eight bits, you can have up to 256 elements in the enumeration type.

If the enumeration type is used as a component type in an array or record definition that is further constrained by a pragma PACK or a record representation clause, the size of this component will be determined by the pragma PACK or the record representation clause. This allows the array or record type to temporarily override any size specification that may have applied to the enumeration type.

The Ada/800 compiler provides a complete implementation of size specifications. Nevertheless, because enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

### Alignment of an Enumeration Type

An enumeration type is byte aligned if the size of the type is less than or equal to eight bits. An enumeration type is aligned on a 2-byte boundary (16 bit or half-word aligned) if the size of the type is less than or equal to 16 bits. An enumeration type is aligned on a 4-byte boundary (32 bit or word aligned) if the size of the type is less than or equal to 32 bits.

## F 4.2 Integer Types

### Predefined Integer Types

The HP 9000 Series 800 Computer System provides these three predefined integer types:

```

type SHORT_SHORT_INTEGER
    is range -(2**7) .. (2**7)-1;    -- 8-bit signed
type SHORT_INTEGER
    is range -(2**15) .. (2**15)-1;  -- 16-bit signed
type INTEGER
    is range -(2**31) .. (2**31)-1;  -- 32-bit signed

```

An integer type declared by a declaration of the form

```
type T is range L .. U;
```

is implicitly derived from a predefined integer type. The compiler automatically selects the smallest predefined integer type whose range contains the values L to U, inclusive.

### Internal Codes of Integer Values

The internal codes for integer values are represented using the two's complement binary method. The compiler does not ever represent integer values using any kind of a bias representation. Thus, one internal code will always represent the same literal value for any Ada integer type.

### Minimum Size of an Integer Type or Subtype

The minimum size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype.

A static subtype of a null range has a minimum size of one. Otherwise, define  $m$  and  $M$  to be the smallest and largest values for the internal codes values of the subtype. The minimum size  $L$  is determined as follows:

Value of $m$	Calculation of $L$ - smallest positive integer such that:	Representation
$m \geq 0$	$M \leq (2^{**}L) - 1$	Unsigned
$m < 0$	$-(2^{**}(L-1)) \leq m$ and $M \leq (2^{**}(L-1))-1$	Signed two's complement



**Example**

```

type MY_INT is range 0 .. 31;
-- The minimum size of MY_INT is 5 bits using
-- an unsigned representation

subtype SOME_INT is MY_INT range 5 .. 7;
-- The minimum size of SOME_INT is 3 bits.
-- The internal representation of 7 requires three
-- binary bits using an unsigned representation.

subtype DYNAMIC_INT is MY_INT range L .. U;
-- Assuming that L and U are dynamic, (i.e. not known at compile time)
-- The minimum size of DYNAMIC_INT is the same as its base type,
-- MY_INT, which is 5 bits.

type ALT_INT is range -1 .. 16;
-- The minimum size of MY_INT is 6 bits,
-- because using a 5-bit signed integer we
-- can only represent numbers in the range -16 .. 15
-- and using a 6-bit signed integer we
-- can represent numbers in the range -32 .. 31
-- Since we must represent 16 as well as -1 the
-- compiler must choose a 6-bit signed representation

```

**Size of an Integer Type**

The sizes of the predefined integer types `SHORT_SHORT_INTEGER`, `SHORT_INTEGER` and `INTEGER` are 8, 16 and 32 bits, respectively.

When no size specification is applied to an integer type, the default size is that of the predefined integer type from which it derives, directly or indirectly.

**Example**

```

type S is range 80 .. 100;
-- Type S is derived from SHORT_SHORT_INTEGER
-- its default size is 8 bits.

type M is range 0 .. 255;
-- Type M is derived from SHORT_INTEGER
-- its default size is 16 bits.

type Z is new M range 80 .. 100;
-- Type Z is indirectly derived from SHORT_INTEGER
-- its default size is 16 bits.

type L is range 0 .. 99999;
-- Type L is derived from INTEGER
-- its default size is 32 bits.

```

## Implementation-Dependent Characteristics

```
type UNSIGNED_BYTE is range 0 .. (2**8)-1;
for UNSIGNED_BYTE'SIZE use 8;
-- Type UNSIGNED_BYTE is derived from SHORT_INTEGER
-- its actual size is 8 bits.

type UNSIGNED_HALFWORD is range 0 .. (2**16)-1;
for UNSIGNED_HALFWORD'SIZE use 16;
-- Type UNSIGNED_HALFWORD is derived from INTEGER
-- its actual size is 16 bits.
```

When a size specification is applied to an integer type, this integer type and all of its subtypes have the size specified by the length clause. The size specification must specify a value greater than or equal to the minimum size of the type. If the size specification specifies that the minimum size and the lower bound of the range is not negative, the internal representation will be unsigned. Thus, when using a size specification of eight bits, you can represent an integer range from 0 to 255.

Using a size specification on an integer type allows you to define unsigned machine integer types. The compiler fully supports unsigned machine integer types that are either 8 or 16 bits. The 8-bit unsigned machine integer type is derived from the 16-bit predefined type `SHORT_INTEGER`. Using the 8-bit unsigned integer type in an expression results in it being converted to the predefined 16-bit signed type for use in the expression. This same method also applies to the 16-bit unsigned machine integer type, such that using the type in an expression results in a conversion to the predefined 32-bit signed type.

However, Ada does not allow the definition of an unsigned integer type that has a greater range than the largest predefined integer type. `INTEGER` is the largest predefined integer type and is represented as a 32-bit signed machine integer. Because the Ada language requires predefined integer types to be symmetric about zero (*Ada RM*, Section 3.5.4), it is not possible to define a 32-bit unsigned machine integer type, because the largest predefined integer type, `INTEGER`, is also a 32-bit type.

If the integer type is used as a component type in an array or record definition that is further constrained by a pragma `PACK` or record representation clause, the size of this component will be determined by the pragma `PACK` or record representation clause. This allows the array or record type to temporarily override any size specification that may have applied to the integer type.

The Ada/800 compiler provides a complete implementation of size specifications. Nevertheless, because integers are coded using machine integers, the specified length cannot be greater than 32 bits.

### Alignment of an Integer Type

An integer type is byte aligned if the size of the type is less than or equal to eight bits. An integer type is aligned on a 2-byte boundary (16 bit or half-word aligned) if the size of the type is in the range of 9..16 bits. An integer type is aligned on a 4-byte boundary (32 bit or word aligned) if the size of the type is in the range of 17..32 bits.

### Performance of an Integer Type

The type `INTEGER` is the most efficient of the integer types in Ada/800 because the hardware can access these integers and perform overflow checks on them with no additional cost. For the smaller integer types (`SHORT_INTEGER` and `SHORT_SHORT_INTEGER`), the compiler must emit additional instructions for access and overflow checking which increase both the execution time and the size of the generated code.

## F 4.3 Floating Point Types

### Predefined Floating Point Types

The HP 9000 Series 800 Computer System provides two predefined floating point types.

```

type FLOAT is digits 6 range
    -(2.0 - 2.0**(-23))*(2.0**127) ..
    +(2.0 - 2.0**(-23))*(2.0**127);
-- This expresses the decimal range -3.40282E+38 .. 3.40282E+38

type LONG_FLOAT is digits 15 range
    -(2.0 - 2.0**(-52))*(2.0**1023) ..
    +(2.0 - 2.0**(-52))*(2.0**1023);
-- This expresses the decimal range:
-- -1.797693134862315E+308 .. +1.797693134862315E+308

```

A floating point type declared by a declaration of the form

```
type T is digits D [range L .. U];
```

is implicitly derived from a predefined floating point type. The compiler automatically selects the smaller of the two predefined floating point types `FLOAT` or `LONG_FLOAT`, whose number of digits is greater than or equal to `D` and which contains the values `L` to `U` inclusive.

### Internal Codes of Floating Point Values

The internal codes for floating point values are represented using the IEEE standard formats for single precision and double precision floats.

The values of the predefined type `FLOAT` are represented using the single precision float format. The values of the predefined type `LONG_FLOAT` are represented using the double precision float format. The values of any other floating point type are represented in the same way as the values of the predefined type from which it derives, directly or indirectly.

## **Implementation-Dependent Characteristics**

### **Minimum Size of a Floating Point Type or Subtype**

The minimum size of a floating point subtype is 32 bits if its base type is `FLOAT` or a type derived from `FLOAT`; it is 64 bits if its base type is `LONG_FLOAT` or a type derived from `LONG_FLOAT`.

### **Size of a Floating Point Type**

The only size that can be specified for a floating point type in a size specification is its default size (32 or 64 bits)

### **Alignment of a Floating Point Type**

A floating point type `FLOAT` is aligned on a 4-byte boundary (32 bit or word aligned). The floating point type `LONG_FLOAT` is aligned on an 8-byte boundary (64 bit or double-word aligned).

## F 4.4 Fixed Point Types

### Predefined Fixed Point Types

To implement fixed point types, the HP 9000 Series 800 Computer System provides a set of three anonymous predefined fixed point types of the form:

```

type SHORT_FIXED is delta D range
    -(2**7)*SMALL .. +((2**7)-1)*SMALL;
for SHORT_FIXED'SMALL use SMALL;
for SHORT_FIXED'SIZE use 8;

type FIXED is delta D range
    -(2**15)*SMALL .. +((2**15)-1)*SMALL;
for FIXED'SMALL use SMALL;
for FIXED'SIZE use 16;

type LONG_FIXED is delta D range
    -(2**31)*SMALL .. +((2**31)-1)*SMALL;
for LONG_FIXED'SMALL use SMALL;
for LONG_FIXED'SIZE use 32;

-- In the above type definitions SMALL is the largest power of
-- two that is less than or equal to D.
```

A fixed point type declared by a declaration of the form

```
type T is delta D range L .. U;
```

is implicitly derived from one of the predefined fixed point types. The compiler automatically selects the smallest predefined fixed point type using the following method:

1. Choose the largest power of two that is not greater than the value specified for the delta, to use as SMALL.
2. Determine the ranges for the three predefined fixed point types using the value obtained for SMALL.
3. Select the smallest predefined fixed point type whose range contains the values L+SMALL to U-SMALL inclusive.

Using the above method, it is possible that the values L and U lie outside the range of the compiler selected fixed point type. For this reason, the values used in a fixed point range constraint should be expressed as follows, to guarantee that the values of L and U are representable in the resulting fixed point type:

```

type ANY_FIXED is delta D range L-D .. U+D;
-- The values of L and U are guaranteed to be
-- representable in the type ANY_FIXED.
```

## Implementation-Dependent Characteristics

### Internal Codes of Fixed Point Values

The internal codes for fixed point values are represented using the two's complement binary method, as integer multiples of 'SMALL. The value of a fixed point object is 'SMALL multiplied by the stored internal code.

### Small of a Fixed Point Type

The Ada/800 compiler requires that the value assigned to 'SMALL is always a power of two. Ada/800 does not support a length clause that specifies a 'SMALL for a fixed point type that is not a power of two.

If a fixed point type does not have a length clause that specifies the value to use for 'SMALL, the value of 'SMALL is determined by the compiler according to the rules in the *Ada RM*, Section 3.5.9.

### Minimum Size of a Fixed Point Type or Subtype

The minimum size of a fixed point subtype is the minimum number of binary digits that is necessary to represent the values in the range of the subtype using the 'SMALL of the base type.

A static subtype of a null range has a minimum size of one. Otherwise, define  $s$  and  $S$  to be the bounds of the subtype, define  $m$  and  $M$  to be the smallest and greatest model numbers of the base type, and let  $i$  and  $I$  be the integer representations for the model numbers  $m$  and  $M$ . The following axioms hold:

$$\begin{aligned} s &\leq m < M \leq S \\ m - T'BASE'SMALL &\leq s \\ M + T'BASE'SMALL &\geq S \\ M &= T'BASE'LARGE \\ i &= m / T'BASE'SMALL \\ I &= M / T'BASE'SMALL \end{aligned}$$

The minimum size  $L$  is determined as follows:

Value of $i$	Calculation of $L$ - smallest positive integer such that:	Representation
$i \geq 0$	$I \leq (2^{**}L) - 1$	Unsigned
$i < 0$	$-(2^{**}(L-1)) \leq i$ and $I \leq (2^{**}(L-1))-1$	Signed two's complement

**Example**

```

type UF is delta 0.1 range 0.0 .. 100.0;
-- The value used for 'SMALL is 0.0625
-- The minimum size of UF is 11 bits,
-- seven bits before the decimal point
-- four bits after the decimal point
-- and no bits for the sign.

type SF is delta 16.0 range -400.0 .. 400.0;
-- The minimum size of SF is 6 bits,
-- nine bits to represent the range 0 to 511
-- less four bits by the implied decimal point of 16.0
-- and one bit for the sign.

subtype UFS is UF delta 4.0 range 0.0 .. 31.0;
-- The minimum size of UFS is 9 bits,
-- five bits to represent the range 0 to 31
-- four bits for the small of 0.0625 from the base type
-- and no bits for the sign.

subtype SFD is SF range X .. Y;
-- Assuming that X and Y are not static, the minimum size
-- of SFD is 6 bits. (the same as its base type)

```

**Size of a Fixed Point Type**

The sizes of the anonymous predefined fixed point types *SHORT\_FIXED*, *FIXED*, and *LONG\_FIXED* are 8, 16 and 32 bits, respectively.

When no size specification is applied to a fixed point type, the default size is that of the predefined fixed point type from which it derives, directly or indirectly.

## Implementation-Dependent Characteristics

### Example

```
type Q is delta 0.01 range 0.00 .. 1.00;  
-- Type Q is derived from an 8-bit predefined fixed point type,  
-- its default size is 8 bits.  
  
type R is delta 0.01 range 0.00 .. 2.00;  
-- Type R is derived from a 16-bit predefined fixed point type,  
-- its default size is 16 bits.  
  
type S is new R range 0.00 .. 1.00;  
-- Type S is indirectly derived from a 16-bit predefined fixed point type,  
-- its default size is 16 bits.  
  
type SF is delta 16.0 range -400.0 .. 400.0;  
for SF'SIZE use 6;  
-- Type SF is derived from an 8-bit predefined fixed point type,  
-- its actual size is 6 bits.  
  
type UF is delta 0.1 range 0.0 .. 100.0;  
for UF'SIZE use 11;  
-- Type UF is derived from a 16-bit predefined fixed point type,  
-- its actual size is 11 bits.  
-- The value used for 'SMALL is 0.0625
```

When a size specification is applied to a fixed point type, this fixed point type and all of its subtypes have the size specified by the length clause. The size specification must specify a value greater than or equal to the minimum size of the type. If the size specification specifies the minimum size and the lower bound of the range is not negative, the internal representation will be that of an unsigned type.

If the fixed point type is used as a component type in an array or record definition that is further constrained by a pragma PACK or record representation clause, the size of this component will be determined by the pragma PACK or record representation clause. This allows the array or record type to temporarily override any size specification that may have applied to the fixed point type.

The Ada/800 compiler provides a complete implementation of size specifications. Nevertheless, because fixed point objects are coded using machine integers, the specified length cannot be greater than 32 bits.

### Alignment of a Fixed Point Type

A fixed point type is byte aligned if the size of the type is less than or equal to eight bits. A fixed point type is aligned on a 2-byte boundary (16 bit or half-word aligned) if the size of the type is in the range of 9..16 bits. A fixed point type is aligned on a 4-byte boundary (32 bit or word aligned) if the size of the type is in the range of 17..32 bits.



## F 4.5 Access Types

### Internal Codes of Access Values

In the program generated by the compiler, access values are represented using 32-bit machine addresses. The predefined generic function `UNCHECKED_CONVERSION` can be used to convert the internal representation of an access value into any other 32-bit type. You can also use `UNCHECKED_CONVERSION` to assign any 32-bit value into an access value. When interfacing with externally supplied data structures, it may be necessary to use the generic function `UNCHECKED_CONVERSION` to convert a value of the type `SYSTEM.ADDRESS` into the internal representation of an access value. Programs which use `UNCHECKED_CONVERSION` in this manner *cannot* be considered portable across different implementations of Ada.

### Collection Size for Access Types

A length clause that specifies the collection size is allowed for an access type. This collection size applies to all objects of this type and any type derived from this type, as well as any and all subtypes of these types. Thus, a length clause that specifies the collection size is only allowed for the original base type definition and not for any subtype or derived type of the base type.

When no specification of collection size applies to an access type, the attribute `STORAGE_SIZE` returns zero. In this case the compiler will dynamically manage the storage for the access type and it is not possible to determine directly the amount of storage available in the collection for the access type.

The recommended format of a collection size length clause is:

```
U_NUM: constant := 50;      -- The maximum number of elements needed
U_SIZE: constant := U_size; -- Substitute U'SIZE here
--
-- The constant U_SIZE should also be:
-- 1. a multiple of two
-- 2. greater than or equal to four
--
-- Additionally, the type U must have a static size
--
type P is access U; -- Type U is any non-dynamic user defined type.
for P'STORAGE_SIZE use (U_SIZE*U_NUM)+4;
```

In the above example we have specified a collection size that is large enough to contain 50 objects of the type `U`. There is a constant overhead of four bytes for each storage collection. Because the collection manager rounds the element size to be a multiple of two that is four or greater, you must ensure that `U_SIZE` is the smallest multiple of two that is greater than or equal to `U'SIZE` and is greater than or equal to four.

You can also provide a length clause that specifies the collection size for a type that has a dynamic size. It is only possible to specify an upper limit on the amount of memory that can be used by all instances of objects that are of this dynamic type. Because the size is dynamic, you cannot specify the number of elements in the collection.

## **Implementation-Dependent Characteristics**

### **Minimum Size of an Access Type or Subtype**

The minimum size of an access type is always 32 bits.

### **Size of an Access Type**

The size of an access type is 32 bits, the same as its minimum size.

The only size that can be specified for an access type in a size specification clause is its usual size (32 bits).

### **Alignment of an Access Type**

An access type is aligned on a 4-byte boundary (32 bit or word aligned).

## F 4.6 Task Types

### Internal Codes of Task Values

In the program generated by the compiler, task type objects are represented using 32-bit machine addresses.

### Storage for a Task Activation

The attribute `'STORAGE_SIZE` can be used in a length clause to specify the amount of stack space to be reserved for the task type. Each task type object has a fixed size private task data section of 3216 bytes, which contains information that is used by the Ada/800 tasking runtime.

When a length clause is *not* used to specify the amount of stack space, this private task data section *is* included in the value returned by the attribute `'STORAGE_SIZE`. Thus, for task type with a default 8K stack, the attribute `'STORAGE_SIZE` returns 11408.

When a length *is* used to specify the amount of stack space, this private task data section is *not* included in the value returned by the attribute `'STORAGE_SIZE`. The value supplied in the length clause is used to specify the amount of stack space that is allocated to the task object. Thus, each task will have a 3216 byte private data section allocated for the task, in addition to the task stack space. The attribute `'STORAGE_SIZE` will only return the amount of stack space allocated.

In specifying a `'STORAGE_SIZE` for a task type, account must be taken of the stack requirements of the Ada code as well as any interface or Ada runtime code that it calls. Each subprogram requires 48 bytes of stack space for the frame marker and fixed argument area. Additional stack space is required for local and temporary data, as well as for additional arguments. Alignment requirements for data must also be considered when determining the amount of stack space required.

### Minimum Size of a Task Stack

The task object will use 150 bytes of stack space in the first stack frame. Some additional stack space is required to make calls into the Ada runtime. The smallest value that can be safely used for a task with minimal stack needs is approximately 400 bytes. If the task object has local variables or if it makes calls to other subprograms, the stack storage requirements will be larger. The actual amount of stack space used by a task will need to be determined by trial and error. If a tasking program raises `STORAGE_ERROR` or behaves abnormally, you should increase the stack space for the tasks.

### Limitation on Length Clause for Derived Task Types

This storage size applies to all task objects of this type and any task type derived from this type. Thus, a length clause that specifies the storage size is only allowed for the original task type definition and not for any derived task type.

## **Implementation-Dependent Characteristics**

### **Minimum Size of a Task Type or Subtype**

The minimum size of a task type is always 32 bits.

### **Size of a Task Type**

The size of a task type is 32 bits, the same as its minimum size.

The only size that can be specified for a task type in a size specification clause is its usual size (32 bits).

### **Alignment of a Task Type**

A task type is aligned on a 4-byte boundary (32 bit or word aligned).

## F 4.7 Array Types

### Layout of an Array

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last component). All the gaps are the same size, as shown below.



### Array component size and pragma PACK

If the array is not packed, the size of each component is the size of the component type. This size is the default size of the component type unless a size specification applies to the component type.

If the array is packed and the array component type is neither a record or array type, the size of the component is the minimum size of the component type. The minimum size of the component type is used even if a size specification applies to the component type.

Packing the array has no effect on the size of the components when the component type is a record or array type.

### Example

```

type A is array(1..8) of BOOLEAN;
-- The component size of A is the default size
-- of the type BOOLEAN: 8 bits.

type B is array(1..8) of BOOLEAN;
pragma PACK(B);
-- The component size of B is the minimum size
-- of the type BOOLEAN: 1 bit.

type DECIMAL_DIGIT is range 0..9;
-- The default size for DECIMAL_DIGIT is 8 bits
-- The minimum size for DECIMAL_DIGIT is 4 bits

type BCD_NOT_PACKED is array(1..8) of DECIMAL_DIGIT;
-- The component size of BCD_NOT_PACKED is the default
-- size of the type DECIMAL_DIGIT: 8 bits.

type BCD_PACKED is array(1..8) of DECIMAL_DIGIT;
pragma PACK(BCD_PACKED);
-- The component size of BCD_PACKED is the minimum
-- size of the type DECIMAL_DIGIT: 4 bits.

```

**Array Gap Size and Pragma PACK**

If the array type is not packed and the component type is a record type without a size specification clause, the compiler may choose a representation for the array with a gap after each component. Inserting gaps optimizes access to the array components. The size of the gap is chosen so that each array component begins on an alignment boundary.

If the array type is packed, the compiler will generally not insert a gap between the array components. In such cases, access to array components can be slower because the array components will not always be aligned correctly. However, in the specific case where the component type is a record and the record has a record representation clause specifying an alignment, the alignment will be honored and gaps may be inserted in the packed array type.

**Example**

```

type R is
  record
    K : INTEGER;    -- Type Integer is word aligned.
    B : BOOLEAN;    -- Type Boolean is byte aligned.
  end record;
-- Record type R is word aligned. Its size is 40 bits.

type A is array(1..10) of R;
-- A gap of three bytes is inserted after each array component in
-- order to respect the alignment of type R.
-- The size of array type A is 640 bits.

type PA is array(1..10) of R;
pragma PACK(PA);
-- There are no gaps in an array of type PA because
-- of the pragma PACK statement on type PA.
-- The size of array type PA is 400 bits.

type NR is new R;
for NR'SIZE use 40;

type B is array(1..10) of NR;
-- There are no gaps in an array of type B because
-- of the size specification clause on type NR.
-- The size of array type B is 400 bits.

```

### Size of an Array Type or Subtype

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the component and the size of the gap.

The size of an array type or subtype cannot be computed at compile time if any of the following are true:

- if the array has non-static constraints or if it is an unconstrained type with non-static index subtypes (because the number of components can then only be determined at run time)
- if the components are records or arrays and their constraints or the constraints of their subcomponents are not static (because the size of the components and the size of the gaps can then only be determined at run time). Pragma PACK is not allowed in this case.

As indicated above, the effect of a pragma PACK on an array type is to suppress the gaps and to reduce the size of the components, if possible. The consequence of packing an array type is thus to reduce its size.

Array packing is fully implemented by the Ada/800 compiler with this limitation: if the components of an array type are records or arrays and their constraints or the constraints of their subcomponents are not static, the compiler ignores any pragma PACK statement applied to the array type and issues a warning message.

A size specification applied to an array type has no effect. The only size that the compiler will accept in such a length clause is the usual size. Nevertheless, such a length clause can be used to verify that the layout of an array is as expected by the application.

### Alignment of an Array Type

If no pragma PACK applies to an array type and no size specification applies to the component type, the array type is aligned on a 4-byte boundary (32 bit or word aligned).

If a pragma PACK applies to an array type or if a size specification applies to the component type (so that there are no gaps), the alignment of the array type is as given in Table F-3.

Table F-3. Alignment and Pragma PACK

Component (Normal Alignment)	Component (Alignment in Structure)				
	Double-Word	Word	Half-Word	Byte	Bit
Double-Word	Double-Word	Word	Half-Word	Byte	Bit
Word	Word	Word	Half-Word	Byte	Bit
Half-Word	Half-Word	Half-Word	Half-Word	Byte	Bit
Byte	Byte	Byte	Byte	Byte	Bit
Bit	Bit	Bit	Bit	Bit	Bit

## F 4.8 Record Types

### Syntax (record representation clause)

```
for record-type-name use
  record [ alignment-clause ]
    [ component-clause ]
    ...
  end record;
```

### Syntax (alignment clause)

```
at mod static-expression
```

### Syntax (component clause)

```
record-component-name at static-expression
  range static-expression .. static-expression ;
```

### Layout of a Record

A record is allocated in a contiguous area of storage units. The size of a record depends on the size of its components and the size of any gaps between the components. The compiler may add additional components to the record. These components are called implicit components.

The positions and sizes of the components of a record type object can be controlled using a record representation clause as described in the *Ada RM*, Section 13.4. If the record contains compiler generated implicit components, their position also can be controlled using the proper component clause. For more details, see "Implicit Components" in Section 4.8. In the implementation for the HP 9000 Series 800 Computer System, there is no restriction on the position that can be specified for a component of a record. If the component is not a record or an array, its size can be any size from the minimum size to the default size of its base type. If the component is a record or an array, its size must be the size of its base type.



Example (Record with a representation clause):

```

type PSW_BIT is new BOOLEAN;
for PSW_BIT'SIZE use 1;

type CARRY_BORROW is array (1..8) of PSW_BIT;
pragma PACK (CARRY_BORROW);

FIRST_BYTE : constant := 0;
CABO_BYTE  : constant := 1;
THIRD_BYTE : constant := 2;
SYSMASK_BYTE: constant := 3;

type PSW is
  record
    T : PSW_BIT;
    H : PSW_BIT;
    L : PSW_BIT;
    N : PSW_BIT;
    X : PSW_BIT;
    B : PSW_BIT;
    C : PSW_BIT;
    V : PSW_BIT;
    M : PSW_BIT;
    CB : CARRY_BORROW;
    R : PSW_BIT;
    Q : PSW_BIT;
    P : PSW_BIT;
    D : PSW_BIT;
    I : PSW_BIT;
  end record;

-- This type can be used to map the status register of
-- the HP-PA processor.

for PSW use
  record at mod 4;
    T at FIRST_BYTE range 7..7;
    H at SECOND_BYTE range 0..0;
    L at SECOND_BYTE range 1..1;
    N at SECOND_BYTE range 2..2;
    X at SECOND_BYTE range 3..3;
    B at SECOND_BYTE range 4..4;
    C at SECOND_BYTE range 5..5;
    V at SECOND_BYTE range 6..6;
    M at SECOND_BYTE range 7..7;
    CB at CABO_BYTE range 0..7;
    R at SYSMASK_BYTE range 3..3;
    Q at SYSMASK_BYTE range 4..4;
    P at SYSMASK_BYTE range 5..5;
    D at SYSMASK_BYTE range 6..6;
    I at SYSMASK_BYTE range 7..7;
  end record;

```

## Implementation-Dependent Characteristics

In the above example, the record representation clause explicitly tells the compiler both the position and size for each of the record components. The optional alignment clause specifies a 4-byte alignment for this record. In this example every component has a corresponding component clause, although it is not required. If one is not supplied, the choice of the storage place for that component is left to the compiler. If component clauses are given for all components, including any implicit components, the record representation clause completely specifies the representation of the record type and will be obeyed exactly by the compiler.

### Bit Ordering in a Component Clause

The HP Ada compiler for the HP 9000 Series 800 Computer System numbers the bits in a component clause starting from the most significant bit. Thus, bit zero represents the most significant bit of an 8-bit byte and bit seven represents the least significant bit of the byte.

### Value used for SYSTEM.STORAGE\_\_UNIT

The smallest directly addressable unit on the HP 9000 Series 800 Computer System is the 8-bit byte. This is the value used for SYSTEM.STORAGE\_\_UNIT which is implicitly used in a component clause. A component clause specifies an offset and a bit range. The offset in a component clause is measured in units of SYSTEM.STORAGE\_\_UNIT, which for the HP 9000 Series 800 Computer System is an 8-bit byte.

The compiler determines the actual bit address for a record component by combining the byte offset with the bit range. There are several different ways to refer to the same bit address. In the following example, each of the component clauses refer to the same bit address.

#### Example

```
COMPONENT at 0 range 16 .. 18;  
COMPONENT at 1 range 8 .. 10;  
COMPONENT at 2 range 0 .. 2;
```

### Compiler-Chosen Record Layout

If no component clause applies to a component of a record, its size is the size of the base type. Its location in the record layout is chosen by the compiler so as to optimize access to the component. That is, each component of a record follows the natural alignment of the component's base type. Moreover, the compiler chooses the position of the components to reduce the number of gaps or holes in the record and additionally to reduce the size of the record.

Because of these optimizations, there is no connection between the order of the components in a record type declaration and the positions chosen by the compiler for the components in a record object.

## Change in Representation

It is not possible to apply a record representation clause to a derived record type. The compiler will use the same storage representation for all records of the same base type. Thus, the compiler does not support the "Change in Representation" as described in the *Ada RM*, Section 13.6.

## Implicit Components

In some circumstances, access to a record object or to a component of a record object involves computing information that only depends on the discriminant values or on a value that is known only at run time. To avoid unnecessary recomputation, the compiler reserves space in the record to store this information. The compiler will update this information whenever a discriminant on which it depends changes. The compiler uses this information whenever the component that depends on this information is accessed. This information is stored in special components called implicit components. There are three different kinds of implicit components:

- Components that contain an offset value.
- Components that contain information about the record object.
- Components that are descriptors.

Implicit components that contain an offset value from the beginning of the record are used to access indirect components. Implicit components of this kind are called *offset components*. The compiler introduces implicit offset components whenever a record contains indirect components. These implicit components are considered to be declared before any variant part in the record type definition. Implicit components of this kind *cannot* be suppressed by using the pragma `IMPROVE`.

Implicit components that contain information about the record object are used when the record object or component of a record object is accessed. Implicit components of this kind are used to make references to the record object or record components more efficient. These implicit components are considered to be declared before any variant part in the record type definition. There are two implicit components of this kind: `RECORD_SIZE` and `VARIANT_INDEX`. Implicit components of this kind *can* be suppressed by using the pragma `IMPROVE`.

The third kind of implicit components are descriptors that are used when accessing a record component. The implicit component exists whenever the record has an array or record component which depends on a discriminant of the record. An implicit component of this kind is considered to be declared immediately before the record component which it is associated with. There are two implicit components of this kind: `ARRAY_DESCRIPTOR` and `RECORD_DESCRIPTOR`. Implicit components of this kind *cannot* be suppressed by using the pragma `IMPROVE`.

<b>NOTE</b>
-------------

The `-S` option (Assembly Option) to the `ada(1)` command is useful for finding out what implicit components are associated with the record type. This option will detail the exact representation for all record types defined in a compilation unit.

## Implementation-Dependent Characteristics

### Indirect Components

If the offset of a component cannot be computed at compile time, the compiler will reserve space in the record for the computed offset. The compiler computes the value to be stored in this offset at run time. A component that depends on a run time computed offset is said to be an indirect component, while other components are said to be direct.

A pictorial example of a record layout with an indirect component is shown below.

Record Offset	Component Name	Size of Component
0	Component A	16 bits
2	Offset for Component D	16 bits
4	Component B	32 bits
6		
8	Component C	<Size known at run time>
10		
12		
14		
16		
.		
.	Storage for Component D	<Size known at run time>
.		

In the above example, the component D has an offset that cannot be computed at compile time. The compiler then will reserve space in the record to store the computed offset and will store this offset at run time. The other components ( A, B, and C ) are all direct components because their offsets can all be computed at compile time.

### Dynamic Components

If a record component is a record or an array, the size of the component may need to be computed at run time and may depend on the discriminants of the record. These components are called *dynamic components*.

Example (Record with dynamic components):

```

type U_RNG is range 0..255;

type UC_ARRAY is array(U_RNG range <>) of INTEGER;

--
-- The type GRAPH has two dynamic components: X and Y.
--
type GRAPH (X_LEN, Y_LEN: U_RNG) is
  record
    X : UC_ARRAY(1 .. X_LEN); -- The size of X depends on X_LEN
    Y : UC_ARRAY(1 .. Y_LEN); -- The size of Y depends on Y_LEN
  end record;

type DEVICE is (SCREEN, PRINTER);

type COLOR is (GREEN, RED, BLUE);

Q : U_RNG;

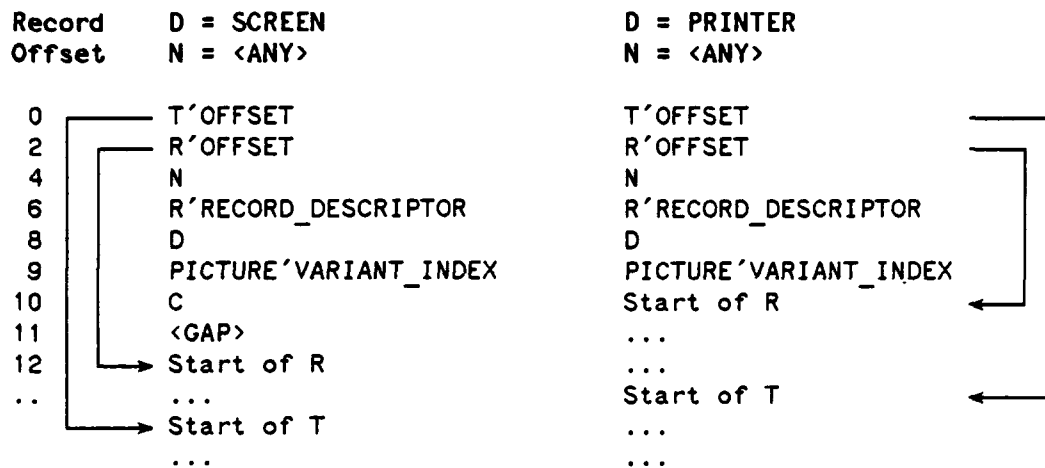
--
-- The type PICTURE has two dynamic components: R and T.
--
type PICTURE (N : U_RNG; D : DEVICE) is
  record
    R : GRAPH(N,N); -- The size of R depends on N
    T : GRAPH(Q,Q); -- The size of T depends on Q
    case D is
      when SCREEN =>
        C : COLOR;
      when PRINTER =>
        null;
    end case;
  end record;

```

Any component that is placed after a dynamic component has an offset that cannot be evaluated at compile time and is thus indirect. To minimize the number of indirect components, the compiler groups the dynamic components and places them at the end of the record. Due to this strategy, the only indirect components are dynamic components. However, all dynamic components are not necessarily indirect. The compiler can usually compute the offset of the first dynamic component and thus it becomes a direct component. Any additional dynamic components are then indirect components.

## Implementation-Dependent Characteristics

A pictorial example of the data layout for the record type PICTURE is shown below.



## Representation of the Offset of an Indirect Component

The offset of an indirect component is always expressed in storage units, which for the HP 9000 Series 800 Computer System are bytes. The space that the compiler reserves for the offset of an indirect component must be large enough to store the maximum potential offset. The compiler will choose the size of an offset component to be either an 8-, 16-, or 32-bit object. It is possible to further reduce the size in bits of this component by specifying it in a component clause.

If C is the name of an indirect component, the offset of this component can be denoted in a component clause by the implementation generated name C'OFFSET.

### Example (Record representation clause for the type GRAPH)

```

for GRAPH use
  record
    X_LEN    at 0 range 0..7;
    Y_LEN    at 1 range 0..7;
    X'OFFSET at 2 range 0..15;
  end record;
--
-- The bit range range for the implicit component
-- X'OFFSET could have been specified as 0..11
-- This would make access to X much slower
--

```

In this example we have used a component clause to specify the location of an offset for a dynamic component. In this example the compiler will choose Y to be the first dynamic component and as such it will have a static offset. The component X will be placed immediately after the end of component Y by the compiler at run time. At run time the compiler will store the offset of this location in the field X'OFFSET. Any references to X will have additional code to compute the run time address of X, using the X'OFFSET field. References to Y will be direct references.

### The Implicit Component RECORD\_SIZE

This implicit component is created by the compiler whenever a record with discriminants has a variant part and the discriminant that defines the variant part has a default expression (that is, a record type that possibly could be unconstrained.) The component 'RECORD\_SIZE contains the size of the storage space required to represent the current variant of the record object. Note that the actual storage allocated for the record object may be more than this.

The value of a RECORD\_SIZE component may denote a number of bits or a number of storage units (bytes). In most cases it denotes a number of storage units (bytes), but if any component clause specifies that a component of the record type has an offset or a size that cannot be expressed using storage units, the value designates a number of bits.

The implicit component RECORD\_SIZE must be large enough to store the maximum size that the record type can attain. The compiler evaluates this size, calls it MS, and considers the type of RECORD\_SIZE to be an anonymous integer type whose range is 0 .. MS.

If R is the name of a record type, this implicit component can be denoted in a component clause by the implementation generated name R'RECORD\_SIZE.

### The Implicit Component VARIANT\_INDEX

This implicit component is created by the compiler whenever the record type has a variant part. It indicates the set of components that are present in a record object. It is used when a discriminant check is to be done.

Within a variant part of a record type, the compiler numbers component lists that themselves do not contain a variant part. These numbers are the possible values for the implicit component VARIANT\_INDEX. The compiler uses this number to determine which components of the variant record are currently valid.

Example (Record with a variant part):

```

type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

type DESCRIPTION( KIND : VEHICLE := CAR) is
  record
    SPEED : INTEGER;
    case KIND is
      when AIRCRAFT | CAR =>
        WHEELS : INTEGER;
        case KIND is
          when AIRCRAFT =>
            WINGSPAN : INTEGER;
          when others =>
            null;
        end case;
      when BOAT =>
        STEAM : BOOLEAN;
      when ROCKET =>
        STAGES : INTEGER;
    end case;
  end record;

```

-- VARIANT\_INDEX is 1

-- VARIANT\_INDEX is 2

-- VARIANT\_INDEX is 3

-- VARIANT\_INDEX is 4

## Implementation-Dependent Characteristics

In the above example, the value of the variant index indicates which of the components are present in the record object; these components are summarized in the table below.

Variant Index	Legal Components
1	KIND, SPEED, WHEELS, WINGSPAN
2	KIND, SPEED, WHEELS
3	KIND, SPEED, STEAM
4	KIND, SPEED, STAGES

The implicit component `VARIANT_INDEX` must be large enough to store the number of component lists that do not contain variant parts. The compiler evaluates this size, calls it `VS`, and considers the type of `VARIANT_INDEX` to be an anonymous integer type whose range is `0 .. VS`.

If `R` is the name of a record type, this implicit component can be denoted in a component clause by the implementation generated name `R'VARIANT_INDEX`.

### The Implicit Component `ARRAY_DESCRIPTOR`

An implicit component of this kind is associated by the compiler with each record component whose type is an array that has bounds that depend on a discriminant of the record.

The structure and contents of the implicit component `ARRAY_DESCRIPTOR` are not described in this manual. Nevertheless, if you are interested in specifying the location of a component of this kind in a component clause, you can obtain the size of the component by supplying the `-S` option (Assembly Option) to the `ada(1)` command.

If `C` is the name of a record component that conforms to the above definition, this implicit component can be denoted in a component clause by the implementation generated name `C'ARRAY_DESCRIPTOR`.

### The Implicit Component `RECORD_DESCRIPTOR`

An implicit component of this kind may be associated by the compiler when a record component is a record type that has components whose size depends on a discriminant of the outer record.

The structure and content of the implicit component `RECORD_DESCRIPTOR` are not described in this manual. Nevertheless, if you are interested in specifying the location of a component of this kind in a component clause, you can obtain the size of the component by applying the `-S` option (Assembly Option) to the `ada(1)` command.

If `C` is the name of a record component that conforms to the above definition, this implicit component can be denoted in a component clause by the implementation generated name `C'RECORD_DESCRIPTOR`.



### Suppression of Implicit Components

Ada/800 provides the capability of suppressing the implicit components `RECORD_SIZE` and `VARIANT_INDEX` from a record type. This can be done using an implementation defined pragma called `IMPROVE`.

#### Syntax

```
pragma IMPROVE ( TIME | SPACE , [ON =>] record_type_name );
```

The first argument specifies whether `TIME` or `SPACE` is the primary criterion for the choice of representation of the record type that is denoted by the second argument.

If `TIME` is specified, the compiler inserts implicit components as described above. This is the default behavior of the compiler. If `SPACE` is specified, the compiler only inserts a `VARIANT_INDEX` component or a `RECORD_SIZE` component if a component clause for one of these components was supplied. If the record type has no record representation clause, both components will be suppressed. Thus, a record representation clause can be used to keep one implicit component while suppressing the other.

A pragma `IMPROVE` that applies to a given record type can occur anywhere that a record representation clause is allowed for this type.

### Size of a Record Type or Subtype

The compiler generally will round up the size of a record type to a whole number of storage units (bytes). If the record type has a component clause that specifies a record component that cannot be expressed in storage units, the compiler will not round up and instead the record size will be expressed as an exact number of bits.

The size of a constrained record type is obtained by adding the sizes of its components and the sizes of its gaps (if any). The size of a constrained record will not be computed at compile time if:

- the record type has non-static constraints.
- a component is an array or record and its size cannot be computed at compile time (that is, if the component has non-static constraints.)

The size of an unconstrained record type is the largest possible size that the unconstrained record type could assume, given the constraints of the discriminant(s). If the size of any component cannot be evaluated exactly at compile time, the compiler will use the maximum size that the component could possibly assume to compute the size of the unconstrained record type.

A size specification applied to a record type has no effect. The only size that the compiler will accept in such a length clause is the usual size. Nevertheless, such a length clause can be used to verify that the layout of a record is as expected by the application.

## Implementation-Dependent Characteristics

### Size of an Object of a Record Type

A record object of a constrained record type has the same size as its base type.

A record object of an unconstrained record type has the same size as its base type if this size is less than or equal to 8192 bytes. The size of the base type is the largest possible size that the unconstrained record type could assume, given the constraints of the discriminant(s). If the size of the base type is larger than 8192 bytes, the record object only has the size necessary to store its current value. Storage space is then allocated and deallocated dynamically based on the current value of the discriminant or discriminants.

### Alignment of a Record Subtype

When a record type does *not* have a record representation clause, or when a record type has a record representation clause *without* an alignment clause, the record type is word aligned to the maximum alignment required by any component of the record (8, 16, 32, or 64 bits). Any subtypes of a record type also have the same alignment as their base type.

For a record type that has a record representation clause *with* an alignment clause, any subtypes of this record type also obey the alignment clause.

An alignment clause can specify that a record type is byte, half-word, word, or double-word aligned (specified as 1, 2, 4, or 8 bytes). Ada/800 does not support alignments larger than an 8-byte alignment.

## F 5. Names for Predefined Library Units

The following names are used by the HP Ada Development System. Do not use any of these names for your library-level Ada units.

ALSYS_ADA_RUNTIME	}	Not available for use.
HIT		
MATH_EXCEPTIONS *	}	Names that you must avoid if you want access to packages that are provided by Hewlett-Packard.
MATH_LIB		
SYSTEM_ENVIRONMENT *		

The packages whose names are followed by \* are available to be used in your programs. These packages are documented in the *Ada User's Guide*.

## **F 6. Address Clauses**

### **Address Clauses for Objects**

An address clause can be used to specify an address for an object as described in the *Ada RM*, section 13.5. When such a clause applies to an object, no storage is allocated for it in the program generated by the compiler. Storage for the object *must* be allocated for the object outside of the Ada program unit unless the address is a memory mapped hardware address. The Ada program accesses the object by using the address specified in the address clause.

An address clause is not allowed for task objects, nor for unconstrained records whose maximum size can be greater than 8192 bytes.

### **Address Clauses for Subprograms**

Address clauses for subprograms are not implemented in the current version of the Ada/800 compiler.

### **Address Clauses for Task Entries**

Address clauses for task entries are not implemented in the current version of the Ada/800 compiler.

### **Address Clauses for Constants**

Address clauses for constants are not implemented in the current version of the Ada/800 compiler.

### **Address Clauses for Packages**

Address clauses for packages are not implemented in the current version of the Ada/800 compiler.

### **Address Clauses for Tasks**

Address clauses for tasks are not implemented in the current version of the Ada/800 compiler.

## F 7. Restrictions on Unchecked Type Conversions

The following limitations apply to the use of `UNCHECKED_CONVERSION`:

- Unconstrained arrays are not allowed as target types.
- Unconstrained record types without defaulted discriminants are not allowed as target types.
- Access types to unconstrained arrays are not allowed as source or target types.
- If the source and target types are each scalar types, the sizes of the types must be equal.
- If the source and target types are each access types, the sizes of the objects which the types denote must be equal.

If the source and the target types are each scalar or access types or if they are both composite types of the same static size, the effect of the function is to return the operand.

In other cases, the effect of unchecked conversion can be considered as a copy.

### **WARNING**

When you do an `UNCHECKED_CONVERSION` among types whose sizes do not match, the code which is generated copies as many bytes as necessary from the source location to fill the target. If the target is larger than the source, the code copies all of the source plus whatever happens to follow the source. Therefore, an `UNCHECKED_CONVERSION` among types whose sizes do not match can produce meaningless results, or can actually cause a trap and abort the program (if these memory locations do not actually exist).

## F 8. Implementation-Dependent Input-Output Characteristics

This section describes the I/O characteristics of Ada on the HP 9000 Series 800 computer. Ada handles I/O with packages, which are discussed in Section F 8.1. File types are described in Section F 8.1.3 and the FORM parameter is discussed in Section F 8.2.

### F 8.1 Ada I/O Packages for External Files

In Ada, I/O operations are considered to be performed on *objects* of a certain file type rather than directly on external files. An external file is anything external to the program that can produce a value to be read or receive a value to be written. In Ada, values transferred to and from a given file must all be of the same type.

Generally, the term *file object* refers to an Ada file of a certain file type, whereas a physical manifestation is known as an *external file*. An external file is characterized by:

- Its NAME, which is a string defining a legal pathname for an external file on the underlying operating system. HP-UX is the underlying operating system for Ada/800. The rules that govern legal pathnames for external files in Ada programs are the same as those that govern legal pathnames in HP-UX. See Section F 8.1.2 for details.
- Its FORM, which allows you to supply implementation-dependent information about the external file characteristics.

Both NAME and FORM appear explicitly in the Ada CREATE and OPEN procedures. These two procedures perform the association of the Ada file object and the corresponding external file. At the time of this association, a FORM parameter is permitted to specify additional characteristics about the external file.

Ada I/O operations are provided by several predefined standard packages. See the *Ada RM*, Section 14 for more details. Table F-4 describes the standard predefined Ada I/O packages.

Table F-4. Standard Predefined I/O Packages

Package	Description and Ada RM Location
SEQUENTIAL_IO	A generic package for sequential files of a single element type. ( <i>Ada RM</i> , Section 14.2.3)
DIRECT_IO	A generic package for direct (random) access files of a single element type. ( <i>Ada RM</i> , Section 14.2.5)
TEXT_IO	A non-generic package for ASCII text files. ( <i>Ada RM</i> , Section 14.3.10)
IO_EXCEPTIONS	A package that defines the exceptions needed by the above three packages. ( <i>Ada RM</i> , Section 14.5)

The generic package LOW\_LEVEL\_IO is not implemented.

### F 8.1.1 Implementation-Dependent Restrictions on I/O Packages

The upper bound for index values in `DIRECT_IO` and for line, column, and page numbers in `TEXT_IO` is:

`COUNT'LAST = 2**31 - 1`

The upper bound for field widths in `TEXT_IO` is:

`FIELD'LAST = 255`

### F 8.1.2 Correspondence between External Files and HP-UX Files

When Ada I/O operations are performed, data is read from and written to external files. Each external file is implemented as a standard HP-UX file. However, before an external file can be used by an Ada program, it must be associated with a file object belonging to that program. This association is achieved by supplying the name of the file object and the name of the external file to the procedures `CREATE` or `OPEN` of the predefined I/O packages. Once the association has been made, the external file can be read from or written to with the file object. Note that for `SEQUENTIAL_IO` and `DIRECT_IO`, you must first instantiate the generic package to produce a non-generic instance. Then you can use the `CREATE` or `OPEN` procedure of that instance. The example at the end of this section illustrates this instantiation process.

The name of the external file can be either of the following:

- an HP-UX pathname
- a null string (for `CREATE` only)

The exception `USE_ERROR` is raised by the procedure `CREATE` if the specified external file cannot be created. The exception `USE_ERROR` is also raised by the procedure `OPEN` if you have insufficient access rights to the file.

If the name is a null string, the associated external file is a temporary file created using the HP-UX facility `tmpnam(3)`. This external file will cease to exist upon completion of the program.

When using `OPEN` or `CREATE`, the Ada exception `NAME_ERROR` is raised if any path component exceeds 255 characters or if an entire path exceeds 1023 characters. This limit applies to path components and the entire path during or after the resolution of symbolic links and context-dependent files (CDFs).

#### WARNING

The absence of `NAME_ERROR` does not guarantee that the path will be used as given. During and after the resolution of symbolic links and context-dependent files (CDFs), the underlying file system may truncate an excessively long component of the resulting pathname. For example, a fifteen character file name used in an Ada program `OPEN` or `CREATE` call will be silently truncated to fourteen characters without raising `NAME_ERROR` by an HP-UX file system that is configured for "short filenames".

## Implementation-Dependent Characteristics

If an existing external file is specified to the CREATE procedure, the contents of that file will be deleted. The recreated file is left open, as is the case for a newly created file, for later access by the program that made the call to create the file.

### Example

```
-- This example creates a file using the generic package DIRECT_IO.
-- It also demonstrates how to close a file and reopen it using a
-- different file access mode.
--
with DIRECT_IO;
with TEXT_IO;
procedure RTEST is
    --
    -- here we instantiate DIRECT_IO on the type INTEGER
    --
    package INTIO is new DIRECT_IO (INTEGER);

    IFILE : INTIO.FILE_TYPE; -- Define a file object for use in Ada

    IVALUE : INTEGER := 0;    -- Ordinary integer object

begin
    INTIO.CREATE ( FILE => IFILE,          -- Ada file is IFILE
                   MODE => INTIO.OUT_FILE, -- MODE allows WRITE only
                   NAME => "myfile"        -- file name is "myfile"
                 );
    TEXT_IO.PUT_LINE ("Created : " &
                     INTIO.NAME (IFILE) &
                     ", mode is " &
                     INTIO.FILE_MODE'IMAGE(INTIO.MODE (IFILE)) );

    INTIO.WRITE (IFILE, 21);              -- Write the integer 21 to the file

    INTIO.CLOSE ( FILE => IFILE);          -- Close the external file

    TEXT_IO.PUT_LINE("Closed file");

    INTIO.OPEN ( FILE => IFILE,            -- Ada file is IFILE
                MODE => INTIO.INOUT_FILE, -- MODE allows READ and WRITE
                NAME => "myfile"          -- file name is "myfile"
              );
    TEXT_IO.PUT_LINE ("Opened : " &
                     INTIO.NAME (IFILE) &
                     ", mode is " &
                     INTIO.FILE_MODE'IMAGE(INTIO.MODE (IFILE)) );

    INTIO.READ (IFILE, IVALUE);           -- Read the first item

    TEXT_IO.PUT_LINE("Read from file, IVALUE = " & INTEGER'IMAGE(IVALUE));

    INTIO.WRITE (IFILE, 65);              -- Write the integer 65 to the file

    TEXT_IO.PUT_LINE("Added an Integer to : " & INTIO.NAME (IFILE));
```



```

INTIO.RESET ( FILE => IFILE,           -- Set MODE to allow READ only
              MODE => INTIO.IN_FILE,   -- and move to the begining
              );                       -- of the file.
                                      -- (IFILE remains open)

TEXT_IO.PUT_LINE ("Reset : " &
                  INTIO.NAME (IFILE) &
                  ", mode is " &
                  INTIO.FILE_MODE'IMAGE(INTIO.MODE (IFILE)) );

while not INTIO.END_OF_FILE(IFILE) loop
    INTIO.READ (IFILE, IVALUE);
    TEXT_IO.PUT_LINE("Read from file, IVALUE = " &
                     INTEGER'IMAGE(IVALUE) );
end loop;

TEXT_IO.PUT_LINE("At the end of file, IFILE");

INTIO.CLOSE ( FILE => IFILE);
TEXT_IO.PUT_LINE("Close file");

end RTEST;

```

In the example above, the file object is IFILE, the external file name relative to your current working directory is myfile, and the actual rooted path could be /PROJECT/myfile. Error or informational messages from the Ada development system (such as the compiler or tools) may mention the actual rooted path.

<p><b>NOTE</b></p>
--------------------

The Ada/800 development system manages files internally so that names involving symbolic links ( see *ln(1)* ) are mapped back to the actual rooted path. Consequently, when the Ada/800 development system interacts with files involving symbolic links, the actual rooted pathname may be mentioned in informational or error messages rather than the symbolic name.

### F 8.1.3 Standard Implementation of External Files

External files have a number of implementation-dependent characteristics, such as their physical organization and file access rights. It is possible to customize these characteristics through the `FORM` parameter of the `CREATE` and `OPEN` procedures, described fully in Section F 8.2. The default of `FORM` is the null string.

The following subsections describe the Ada/800 implementation of these three types of external files: `SEQUENTIAL_IO`, `DIRECT_IO`, and `TEXT_IO` files.

#### F 8.1.3.1 `SEQUENTIAL_IO` Files

A `SEQUENTIAL_IO` file is a sequence of elements that are transferred in the order of their appearance to or from an external file. Each element in the file contains one object of the type that `SEQUENTIAL_IO` was instantiated on. All objects in the file are of this same type. An object stored in a `SEQUENTIAL_IO` file has exactly the same binary representation as an Ada object in the executable program.

The information placed in a `SEQUENTIAL_IO` file depends on whether the type used in the instantiation is a constrained type or an unconstrained type.

For a `SEQUENTIAL_IO` file instantiated with a constrained type, each element is simply the object. The objects are stored consecutively in the file without separators. For constrained types, the number of bytes occupied by each element is the size of the constrained type and is the same for all elements. Files created using `SEQUENTIAL_IO` on constrained types *can* be accessed as `DIRECT_IO` files at a later time. The representation of both `SEQUENTIAL_IO` and `DIRECT_IO` files are the same when using constrained types.

For a `SEQUENTIAL_IO` file instantiated with an unconstrained type, each element is composed of three parts: the size (in bytes) of the object is stored in the file as a 32-bit integer value, the object, and a few optional unused trailing bytes. These unused trailing bytes will only be appended if the `FORM` parameter, `RECORD_UNIT` was specified in the `CREATE` call. This parameter instructs the Ada runtime to round up the size of each element in the file to be an integral multiple of the `RECORD_UNIT` size. The default value for `RECORD_UNIT` is one byte, which means that unused trailing bytes will *not* be appended. The principle use for the `RECORD_UNIT` parameter is in reading and writing external files that are in formats that already use this convention. Files created using `SEQUENTIAL_IO` on unconstrained types *cannot* be accessed as `DIRECT_IO` files at a later time. The representation of `SEQUENTIAL_IO` and `DIRECT_IO` files are *not* the same when using an unconstrained type. See Section F 8.2.9.2 for more information on file structure.

A `SEQUENTIAL_IO` file can be buffered. Buffering is selected by specifying a non-zero value for the `FORM` parameter, `BUFFER_SIZE`. The I/O performance of an Ada program will be considerably improved if buffering is used. By default, no buffering takes place between the physical external file and the Ada program. See Section F 8.2.4 for details on specifying a file `BUFFER_SIZE`.

#### F 8.1.3.2 `DIRECT_IO` Files

A `DIRECT_IO` file is a set of elements each occupying consecutive positions in a linear order. `DIRECT_IO` files are sometimes referred to as random-access files because an object can be transferred to or from an element at any selected position in the file. The position of an element in a `DIRECT_IO` file is specified by its index, which is a number in the range 1 to  $(2^{*31})-1$  of the subtype `POSITIVE_COUNT`. Each element in the file contains one object of the type that `DIRECT_IO` was instantiated on. All objects in the

file are of this same type. The object stored in a `DIRECT_IO` file has exactly the same binary representation as the Ada object in the executable program.

Elements within a `DIRECT_IO` file *always* have the same size. This requirement allows the Ada runtime to easily and quickly compute the location of any element in a `DIRECT_IO` file.

For a `DIRECT_IO` file instantiated with a constrained type, the number of bytes occupied by each element is the size of the constrained type. Files created using `DIRECT_IO` on constrained types *can* be accessed as `SEQUENTIAL_IO` files at a later time. The representation of both `DIRECT_IO` and `SEQUENTIAL_IO` files are the same when using a constrained type.

For `DIRECT_IO` files instantiated with an unconstrained type, the number of bytes occupied by each element is determined by the `FORM` parameter, `RECORD_SIZE`. All of the unconstrained objects stored in the file must have an actual size that is less than or equal to this size. The exception `DATA_ERROR` is raised if the size of an unconstrained object is larger than this size. Files created using `DIRECT_IO` on unconstrained types *cannot* be accessed as `SEQUENTIAL_IO` files at a later time. The representation of `DIRECT_IO` and `SEQUENTIAL_IO` files are *not* the same when using an unconstrained type. See Section F 8.2.9.2 for more information on file structure.

If the file is created with the default `FORM` parameter attributes (see Section F 8.2), only objects of a constrained type can be written to or read from a `DIRECT_IO` file. Although an instantiation of `DIRECT_IO` is accepted for unconstrained types, the exception `USE_ERROR` is raised on any call to `CREATE` or `OPEN` when the default value of the `FORM` parameter is used. You *must* specify the maximum `RECORD_SIZE` for the unconstrained type.

A `DIRECT_IO` file can be buffered. Buffering is selected by specifying a non-zero value for the `FORM` parameter, `BUFFER_SIZE`. The I/O performance of an Ada program will normally be considerably improved if buffering is used. However, for a `DIRECT_IO` file that is accessed in a random fashion, performance can actually be degraded. The buffer will always reflect a contiguous set of elements in the file and if subsequent I/O requests lie outside of the current buffer, the entire buffer will be updated. This could cause performance to degrade if a large buffer is used and each I/O request requires that the buffer be updated. By default, no buffering takes place between the physical external file and the Ada program. See Section F 8.2.4 for details on specifying a file `BUFFER_SIZE`.

### F 8.1.3.3 `TEXT_IO` Files

A `TEXT_IO` file is used for the input and output of information in readable form. Each `TEXT_IO` file is read or written sequentially as a sequence of characters grouped into lines and as a sequence of lines grouped into pages. All `TEXT_IO` column numbers, line numbers, and page numbers are in the range 1 to  $(2^{31})-1$  of subtype `POSITIVE_COUNT`. The line terminator (end-of-line) is physically represented by the character `ASCII.LF`. The page terminator (end-of-page) is physically represented by a succession of the two characters, `ASCII.LF` and `ASCII.FF`, in that order. The file terminator (end-of-file) is physically represented by the character `ASCII.LF` and is followed by the physical end of file. There is no ASCII character that marks the end of a file. An exception to this rule occurs when reading from a terminal device. In this case, the character `ASCII.EOT` (CTRL-D) is always used by the Ada runtime to indicate the end-of-file. The Ada runtime does not use the EOF character currently defined for the device (as set by `stty(1)`, for example). See Section F 8.2.9.1 in this appendix for more information about the structure of text files.

If you leave the control of line, page, and file terminators to the Ada runtime and use only `TEXT_IO` subprograms to create and modify the text file, you need not be concerned with the above terminator implementation details. However, you *must not* output the characters `ASCII.LF` or `ASCII.FF` when using `TEXT_IO.PUT` operations because these characters would be interpreted as line terminators or as

## Implementation-Dependent Characteristics

page terminators when the file was later read using `TEXT_IO.GET`. If you effect structural control by explicitly outputting these control characters, it is your responsibility to maintain the integrity of the external file.

If your text file was not created using `TEXT_IO`, your text file may not be in a format that can be interpreted correctly by `TEXT_IO`. It may be necessary to filter the file or perform other modifications to the text file before it can be correctly interpreted as an Ada text file. See Section F 8.2.9.1 for information on the structure of `TEXT_IO` files.

The representation of a `TEXT_IO` file is a sequence of ASCII characters. It is possible to use `DIRECT_IO` or `SEQUENTIAL_IO` to read or write a `TEXT_IO` file. The Ada type `CHARACTER` *must* be used in the instantiation of `DIRECT_IO` or `SEQUENTIAL_IO`. It is *not* possible to use `DIRECT_IO` or `SEQUENTIAL_IO` on the Ada type `STRING` to read or write a `TEXT_IO` file.

A `TEXT_IO` file can be buffered. Buffering is selected by specifying a non-zero value for the `FORM` parameter, `BUFFER_SIZE`. The I/O performance of an Ada program will be considerably improved if buffering is used. By default, no buffering takes place between the physical external file and the Ada program. However, terminal input is line buffered by default. See Section F 8.2.4 and Section F 8.2.8 for details.

### F 8.1.4 Default Access Protection of External Files

HP-UX provides protection of a file by means of access rights. These access rights are used within Ada programs to protect external files. There are three levels of protection:

- User (the owner of the file).
- Group (users belonging to the owner's group).
- Others (users belonging to other groups).

For each of these levels, access to the file can be limited to one or several of the following rights: read, write, or execute. The default HP-UX external file access rights are specified by using the `umask(1)` command (see `umask(1)` and `umask(2)` in the *HP-UX Reference*.) Access rights apply equally to sequential, direct, and text files. See Section F 8.2.3 on the `FORM` parameter for information about specifying file permissions at the time of `CREATE`.

### F 8.1.5 System Level Sharing of External Files

Under HP-UX, several programs or processes can access the same HP-UX file simultaneously. Each program or process can access the HP-UX file either for reading or for writing. Although HP-UX can provide file and record locking protection using *fcntl(2)* or *lockf(2)*, Ada/800 does not utilize this feature when it performs I/O on external files. Thus, the external file that Ada reads or writes is not protected from simultaneous access by non-Ada processes, or by another Ada program that is executing concurrently. Such protection is outside the scope of Ada/800. However, you can limit access to a file by specifying a file protection mask using the FORM parameter when you create the file. See Section F 8.2.3 for more information.

The effects of sharing an external file depend on the nature of the file. You must consider the nature of the device attached to the file object and the sequence of I/O operations on the device. You also must consider the effects of file buffering if you are attempting to update a file that is being shared.

For shared files on random access devices, such as disks, the data is shared. Reading from one file object does not affect the file positioning of another file object, nor the data available to it. However, writing to a file object may not cause the external file to be immediately updated; see Section F 8.2.5.1, "Interaction of File Sharing and File Buffering" for details.

For shared files on sequential devices or interactive devices, such as magnetic tapes or keyboards, the data is no longer shared. In other words, a magnetic record or keyboard input character read by one I/O operation is no longer available to the next operation, whether it is performed on the same file object or not. This is simply due to the sequential nature of the device.

By default, file objects represented by STANDARD\_INPUT and STANDARD\_OUTPUT are preconnected to the HP-UX streams stdin and stdout ( see *stdio(5)* ), and thus are of this sequential variety of file. The HP-UX stream stderr is not preconnected to an Ada file but is used by the Ada runtime system for error messages. An Ada subprogram called PUT\_TO\_STANDARD\_ERROR is provided in the package SYSTEM\_ENVIRONMENT which allows your program to output a line to the HP-UX stream stderr.

#### NOTE

The sharing of external files is system-wide and is managed by the HP-UX operating system. Several programs may share one or more external files. The file sharing feature of HP Ada using the FORM parameter SHARED, which is discussed in Section F 8.2.5, is not system-wide, but is a file sharing within a single Ada program and is managed by that program.

### F 8.1.6 I/O Involving Access Types

When an object of an access type is specified as the source or destination of an I/O operation (read or write), the 32-bit binary access value is read or written unchanged. If an access value is read from a file, make sure that the access value read designates a valid object. This *will only* be the case if the access value read was previously written by the *same execution* of the program that is reading it, and the object which it designated at the time it was written still exists (that is, the scope in which it was allocated has not been exited, nor has an UNCHECKED\_DEALLOCATION been performed on it). A program may execute erroneously or raise PROGRAM\_ERROR if an access type read from a file does not designate a valid object. In general, I/O involving access types is strongly discouraged.

### F 8.1.7 I/O Involving Local Area Networks

This section assumes knowledge of both remote file access and networks. It describes Ada program I/O involving these two Local Area Network (LAN) services available on the Series 800 computers:

- RFA systems: remote file access using the NS/9000 network services software.
- NFS<sup>\*</sup> systems: remote file access using the NFS network services software.

The Ada programs discussed here are executed on a local (host) computer. These programs access or create files on a remote system, which is connected to a mass storage device not directly connected to the host computer. The remote file system can be mounted and accessed by the host computer using RFA or NFS LAN services. RFA systems are described in the manuals *Using Network Services* and *Installing and Administering Network Services*. NFS systems are described in the manuals *Using NFS Services* and *Installing and Administering NFS Services*.

Note that Ada I/O can be used reliably across local area networks using RFA only if the network special files representing remote file systems are contained in the directory /net, as is customary on HP-UX systems. See the *HP-UX System Administrator Manual* for more details.

<sup>\*</sup> NFS is a trademark of Sun Microsystems, Inc.

### F 8.1.7.1 RFA Systems

Properly specified external files can be created or accessed reliably from Ada programs across the LAN on RFA systems. You can create or access a file only if an RFA connection exists to the remote file system at the time your Ada program is executed. The example below illustrates remote file access and use.

In this example, a remote network connection to a system *cezanne* is assumed. This connection could have been made by typing the following (where *\$* is the shell prompt):

```
$ netunam /net/cezanne user_name:
Password: user_passwd
$
```

#### Example

```
with DIRECT_IO;
procedure LANTEST is
```

```
  -- instantiate the generic package DIRECT_IO
  -- for files whose component type is INTEGER.
  package TESTIO is new DIRECT_IO(INTEGER);
```

```
  REMOTE_FILE : TESTIO.FILE_TYPE; -- Define a file object for use in Ada.
  WVALUE      : INTEGER := 35;
  RVALUE      : INTEGER := -1;
```

```
begin
```

```
  -- create a remote file
  TESTIO.CREATE (FILE => REMOTE_FILE,
                 MODE => TESTIO.OUT_FILE, -- Access mode allows WRITE
                 NAME => "/net/cezanne/project/test.file");
```

```
  -- Write an integer (35) to the file
  TESTIO.WRITE (REMOTE_FILE, WVALUE);
```

```
  -- Reset the file pointer in the file and change
  -- the access mode to allow READ only
  TESTIO.RESET (REMOTE_FILE, TESTIO.IN_FILE);
```

```
  -- Read from the file, rvalue should now be (35)
  TESTIO.READ (REMOTE_FILE, RVALUE);
```

```
  -- Close the file
  TESTIO.CLOSE (FILE => REMOTE_FILE);
end LANTEST;
```

### F 8.1.7.2 NFS Systems

If an Ada program expects to access or create a file on a remote file system using NFS LAN services, the remote volumes that contain the file system must be mounted on the host computer prior to the execution of the Ada program.

For example, assume that the remote system (cezanne) exports a file system `/project`. `/project` is mounted on the host computer as `/ada/project`. Files in this remote file system are accessed or created by references to the files as if they were part of the local file system. To access the file `test.file`, the program would reference `/ada/project/test.file` on the local system. Note that `test.file` appears as `/project/test.file` on the remote system. The `netunam(1)` command (from HP's NS/9000) is not used in NFS.

The remote file system must be exported to the local system before it can be locally mounted using the `mount(1m)` command.

### F 8.1.8 Potential Problems with I/O From Ada Tasks

In an Ada tasking environment on the HP 9000 Series 800, the Ada runtime must ensure that a file object is protected against attempts to perform multiple simultaneous I/O operations on it. If such protection was not provided, the internal state of the file object could become incorrect. For example, consider the case of two tasks each writing to `STANDARD_OUTPUT` simultaneously. The internal values of a text file object include information returned by `TEXT_IO.COL`, `TEXT_IO.LINE`, and `TEXT_IO.PAGE` functions. These internal values are volatile and any I/O operations that change these values must be completed before any other I/O operations are begun on the file object. Thus, the file object is protected by the Ada runtime for the duration of the I/O operation. If another task is scheduled and runs before the I/O operation has completed and this task attempts to perform I/O on the protected file object, the exception `PROGRAM_ERROR` is generated at the point of the I/O operation. If this exception is not caught by the task, the task will be terminated.

Note that the file protection provided by the Ada runtime is *not* the same as the protection provided by the use of the `SHARED` attribute of the `FORM` parameter of `CREATE` or `OPEN` calls. The `FORM` parameter either prohibits or allows multiple Ada file objects to share the same external file. In contrast, the file protection provided by the Ada runtime prohibits the simultaneous sharing of the *same* Ada file object between tasks. The `SHARED` attribute always deals with *multiple* Ada file objects.

The file protection provided by the Ada runtime will only be a problem when the *same* Ada file object is used by different tasks. When each task uses a separate file object, it is not necessary to provide explicit synchronization when performing I/O operations. This is true even when the file objects are sharing the same external file. However, for this case, you will need to consider the effects of the `SHARED` attribute and/or file buffering.

#### CAUTION

It is your responsibility to utilize proper synchronization and mutual exclusion in the use of shared resources. Note that shared access to a common resource (in this case, a file object) could be achieved by a rendezvous between tasks that share that resource. If you write a program in which two tasks attempt to perform I/O operations on the same logical file without proper synchronization, that program is erroneous. ( See *Ada RM*, Section 9.11 )



### F 8.1.9 I/O Involving Symbolic Links

Some caution must be exercised when using an Ada program that performs I/O operations to files that involve symbolic links. For more detail on the use of symbolic links to files in HP-UX, see *ln(1)*.

Creating a symbolic link to a file creates a new name for that file; that is, an alias for the actual file name is created. If you use the actual file name or its alias (that is, the name involving symbolic links), Ada I/O operations will work correctly. However, the NAME function in the TEXT\_IO, SEQUENTIAL\_IO, and DIRECT\_IO packages will always return the actual rooted path of a file and *not* a path involving symbolic links.

### F 8.1.10 Ada I/O System Dependencies

Ada/800 has a requirement (see *Ada RM*, Section 14.2.1(21)) that the NAME function must return a string that uniquely identifies the external file in HP-UX. In determining the unique file name, the Ada runtime system may need to access directories and directory entries not directly associated with the specified file. This is particularly true when the path to the file specified involves either NFS or NS/9000 RFA remote file systems. This access involves HP-UX operating system calls that are constrained by HP-UX access permissions and are subject to failures of the underlying file system, as well as by network behavior.

#### WARNING

It is during the Ada/800 OPEN and CREATE routines that the unique file name is determined for later use by the NAME function. If it was not possible to determine the unique file name, the exception NAME\_\_ERROR, USE\_\_ERROR, or DEVICE\_\_ERROR (as appropriate for the actual error encountered) is raised by the call to OPEN or CREATE routines. The Ada NAME function will only report this unique file name for the associated file object after a successful call to OPEN or CREATE.

If the underlying file system or network denies access (possibly due to a failed remote file system) or the access permissions are improper, the OPEN or CREATE call will raise an exception or for the case of a network failure, the call might not complete until the situation is corrected.

For example, when opening a file, the Ada exception NAME\_\_ERROR is raised if there are any directories in the rooted path of the file that are not readable or searchable by the "effective uid" of the program. This restriction applies to intermediate path components that are encountered during the resolution of symbolic links.

Also, if access to an NFS "hard" mounted remote file system is lost (possibly due to a network failure), subsequent OPEN or CREATE calls on a file whose actual rooted path contains the parent directory of the NFS mount point might not complete until the NFS failure is corrected (whether or not the actual file being accessed is on the failed NFS volume.)

## F 8.2 The FORM Parameter

For both the CREATE and OPEN procedures in Ada, the FORM parameter specifies the characteristics of the external file involved.

### F 8.2.1 An Overview of FORM Attributes

The FORM parameter is composed from a list of attributes that specify the following:

- File protection
- File buffering
- File sharing
- Appending
- Blocking
- Terminal input
- File structuring
- Terminal Input

### F 8.2.2 The Format of FORM Parameters

Attributes of the FORM parameter have an attribute keyword followed by the Ada "arrow symbol" ( $\Rightarrow$ ), and followed by a qualifier or numeric value.

The arrow symbol and qualifier are not always needed and can be omitted. Thus, the format for an attribute specifier is

*KEYWORD*

or

*KEYWORD  $\Rightarrow$  QUALIFIER*

The general format for the FORM parameter is a string formed from a list of attributes with attributes separated by commas. ( FORM attributes are distinct from Ada attributes and the two are not related. ) The FORM parameter string is *not* case sensitive. The arrow symbol can be separated by spaces from the keyword and qualifier. The two forms below are equivalent:

*KEYWORD  $\Rightarrow$  QUALIFIER*

*KEYWORD $\Rightarrow$ QUALIFIER*

In some cases, an attribute can have multiple qualifiers that can be presented at the same time. In cases that allow multiple qualifiers, additional qualifiers are introduced with an underscore (  ). Note that spaces are not allowed between the additional qualifiers; only underscore characters are allowed. Otherwise, a `USE_ERROR` exception is raised by CREATE. The two examples that follow illustrate the FORM parameter format.

The first example illustrates the use of the FORM parameter in the `TEXT_IO.OPEN` to set the file buffer size.

```

-- Example of opening a file using the non-generic package TEXT_IO.
-- This illustrates the use of the FORM parameter BUFFER_SIZE.
-- Note: "inpt_file" must exist, or NAME_ERROR will be raised.
--
with TEXT_IO;
procedure STEST is

    TFILE : TEXT_IO.FILE_TYPE; --Define a file object for use in Ada

begin -- STEST
    TEXT_IO.OPEN (FILE => TFILE,                -- Ada file is TFILE
                  MODE => TEXT_IO.IN_FILE,      -- Access allows reading
                  NAME => "inpt_file",          -- file name is "inpt_file"
                  FORM => "BUFFER_SIZE => 4096" -- Buffer Size is 4096 bytes
                  );
end STEST;

```

The second example illustrates the use of the FORM parameter in TEXT\_IO.CREATE. This example sets the access rights of the owner (HP-UX file permissions) on the created file and shows multiple qualifiers being presented at the same time.

```

TEXT_IO.CREATE (OUTPUT_FILE, TEXT_IO.OUT_FILE, OUTPUT_FILE_NAME,
                FORM=>"owner=>read_write_execute");

```

### F 8.2.3 The FORM Parameter Attribute - File Protection

The file protection attribute is only meaningful for a call to the CREATE procedure.

File protection involves two independent classifications. The first classification specifies *which user* can access the file and is indicated by the keywords listed in Table F-5.

**Table F-5. User Access Categories**

Category	Grants Access To
OWNER	Only the owner of the created file.
GROUP	Only the members of a defined group.
WORLD	Any other users.

Note that WORLD is similar to "others" in HP-UX terminology, but was used in its place because OTHERS is an Ada reserved word.

## Implementation-Dependent Characteristics

The second classification specifies *access rights* for each classification of user. The four general types of access rights, which are specified in the FORM parameter qualifier string, are listed in Table F-6.

Table F-6. File Access Rights

Category	Allows the User To
READ	Read from the external file.
WRITE	Write to the external file.
EXECUTE	Execute a program stored in the external file.
NONE	The user has no access rights to the external file. (This qualifier overrides any prior privileges).

More than one access right can be specified for a particular file. Additional access rights can be indicated by separating them with an underscore, as noted earlier. The following example using the FORM parameter in TEXT\_IO.CREATE sets access rights of the owner and other users (HP-UX file permissions) on the created file. This example illustrates multiple qualifiers being used to set several permissions at the same time.

```
TEXT_IO.CREATE (OUTPUT_FILE, TEXT_IO.OUT_FILE, OUTPUT_FILE_NAME,  
FORM=>"owner=>read_write_execute, world=>none");
```

Note that the HP-UX command *umask(1)* may have set the default rights for any unspecified permissions. In the previous example, permission for the users in the category GROUP were unspecified. Typically, the default *umask* will be set so that the default allows newly created files to have read and write permission (and no execute permission) for each category of user (USER, GROUP, and WORLD).

Consider the case where the users in WORLD want to execute a program in an external file, but only the owner may modify the file. The appropriate FORM parameter is then:

```
WORLD => EXECUTE,  
OWNER => READ_WRITE_EXECUTE
```

This would be applied as:

```
TEXT_IO.CREATE (OUTPUT_FILE, TEXT_IO.OUT_FILE, OUTPUT_FILE_NAME,  
FORM=>"world=>execute, owner=>read_write_execute");
```

Repetition of the same qualifier within attributes is illegal:

```
WORLD => EXECUTE_EXECUTE          -- NOT legal
```

But repetition of entire attributes is allowed:

```
WORLD => EXECUTE, WORLD => EXECUTE  -- legal
```

### F 8.2.4 The FORM Parameter Attribute - File Buffering

The buffer size can be specified by the attribute:

```
BUFFER_SIZE => size__in__bytes
```

The default value for `BUFFER_SIZE` is 0, which means no buffering. Using the file buffering attribute will improve I/O performance by a considerable amount in most cases. If I/O performance is a concern for disk files, the attribute `BUFFER_SIZE` should be set to a value that is an integral multiple of the size of a physical disk block. The size of a physical disk block can be found in `<sys/param.h>` and is 1024 bytes for the HP 9000 Series 300.

An example of the use of the FORM parameter in the `TEXT_IO.OPEN` to set the file buffer size is shown below:

```
-- An example of creating a file using the non-generic package TEXT_IO.
-- This illustrates the use of the FORM parameter BUFFER_SIZE.

with TEXT_IO;
procedure T_TEST is

    BFILE : TEXT_IO.FILE_TYPE; -- Define a file object for use by Ada

begin -- T_TEST

    TEXT_IO.CREATE (FILE => BFILE,                -- Ada file is BFILE
                    MODE => TEXT_IO.OUT_FILE,      -- MODE is WRITE only
                    NAME => "txt_file",            -- External file "txt_file"
                    FORM => "BUFFER_SIZE => 8192" -- Buffer size is 8192 bytes
                    );

end T_TEST;
```

### F 8.2.5 The FORM Parameter Attribute - File Sharing

The file sharing attribute of the FORM parameter allows you to specify what kind of sharing is permitted when multiple file objects access the same external file. This control over file sharing is *not* system-wide, it is instead limited to a single Ada program. The HP-UX operating system controls file sharing at the system level. See Section F 8.1.5 for information on system level file sharing between separate programs.

An external file can be shared; that is, the external file can be associated simultaneously with several logical file objects created by the `OPEN` or `CREATE` procedures. The file sharing attributes forbids or limits this capability by specifying one of the modes listed in Table F-7.

Table F-7. File Sharing Attribute Modes

Mode	Description
NOT_SHARED	Indicates exclusive access. No other logical file can be associated with the external file.
SHARED=>READERS	Only logical files of mode IN can be associated with the external file.
SHARED=>SINGLE_WRITER	Only logical files of mode IN and at most one file with mode OUT can be associated with the external file.
SHARED=>ANY	No restrictions; this is the default.

A `USE_ERROR` exception is raised if either of the following conditions exists for an external file already associated with at least one logical Ada file:

- The `OPEN` or `CREATE` call specifies a file sharing attribute *different* than the current one in effect for this external file. Remember the attribute `SHARED=>ANY` is provided if the shared attribute is missing from the `FORM` parameter.
- A `RESET` call that changes the `MODE` of the file and violates the conditions imposed by the current file sharing attribute. (that is, if `SHARED=>READERS` is in effect the `RESET` call cannot change a reader into writer)

The current restriction imposed by the file sharing attribute disappears when the last logical file linked to the external file is closed. The next call to `CREATE` or `OPEN` can and does establish a new file sharing attribute for this external file. See Section F 8.1.8 for information about potential problems with I/O from Ada tasks.

#### F 8.2.5.1 Interaction of File Sharing and File Buffering

For files that are *not buffered* (the default), multiple I/O operations on an external file shared by several file objects are processed in the order they occur. Each Ada I/O operation will be translated into the appropriate HP\_UX system call (`read(2)`, `write(2)`, `creat(2)`, `open(2)`, or `close(2)`) and the external file will be updated by the HP-UX I/O runtime. Note that if file access is performed across a network device, the external file may not be immediately updated. However, additional I/O operations on the file will be queued and must wait until the original operation has completed. This allows multiple readers and multiple writers for the external file.

For files that are *buffered*, multiple I/O operations each operate sequentially only within the buffer that is associated with the file object and each file object has its own buffer. For write operations, this buffer is flushed to the disk either when the buffer is full, or when the file index is positioned outside of the buffer, or when the file is closed. The external file only reflects the changes made by a write operation after the buffer is flushed to the disk. Any accesses to the external file that occur before the buffer is flushed will not reflect the changes made to the file that exist only in the buffer.

Due to this behavior, shared files should *not* be buffered if any write operations are to be performed on this file. This would be the case for file objects of the mode `OUT_FILE` or `INOUT_FILE`. Thus, when using buffered files safely, *no* writers are allowed, but multiple readers *are* allowed.

File buffering is enabled by using the `FORM` parameter attributes at the time you open or create the file. If file buffering is enabled for a file, you should also specify a file sharing attribute of either `NOT_SHARED` or `SHARED=>READERS` to prevent the effects of file buffering and file sharing interfering with one another. The Ada runtime will raise the exception `USE_ERROR` if any attempt is made to share the file or to share and write the file, when the above file sharing attributes are provided as `FORM` parameters.

If the possibility of shared access exists in your Ada program for sequential devices or interactive devices, you should specify a file sharing attribute of `NOT_SHARED`. This will prevent the negative effects of file sharing on these kinds of devices.

### F 8.2.6 The `FORM` Parameter - Appending to a File

The `APPEND` attribute can only be used with the procedure `OPEN`. Its format is:

`APPEND`

Any output will be placed at the end of the named external file.

Under normal circumstances, when an external file is opened, an index is set that points to the beginning of the file. If the `APPEND` attribute is present for a sequential or text file, data transfer begins at the end of the file. For a direct access file, the value of the index is set to one more than the number of records in the external file.

The `APPEND` attribute is *not* applicable to terminal devices.

### F 8.2.7 The `FORM` Parameter Attribute - Blocking

This attribute has two alternative forms:

`BLOCKING`

or

`NON_BLOCKING`

#### F 8.2.7.1 Blocking

If the blocking attribute is set, the read or write operation will cause the HP-UX process to block until the read or write request is satisfied. This means that all Ada tasks are blocked from running until the data transfer is complete.

The default for this attribute depends on the actual program. The default is `BLOCKING` for programs without any task declarations and is `NON_BLOCKING` for a program containing tasks. This allows tasking programs to take advantage of their parallelism in the presence of certain I/O requests. There is no advantage in specifying `NON_BLOCKING` for a non-tasking program because the program must wait for the I/O request to complete before continuing its sequential execution.

### F 8.2.7.2 Non-Blocking

The `NON_BLOCKING` attribute specifies that when a read request cannot be immediately satisfied, the Ada runtime should schedule another task to run and retry the read operation later. This attribute is currently only applicable for terminal devices and pipes. In the case of a pipe, a write request may also cause the current task to be rescheduled, and another task will run while the pipe buffer is full. This attribute sets the `O_NDELAY` flag in the HP-UX file descriptor and allows the HP-UX process to continue running if there is no data available to be read from the terminal or pipe.

### F 8.2.8 The FORM Parameter - Terminal Input

The terminal input attribute takes one of two alternative forms:

`TERMINAL_INPUT => LINES,`

`TERMINAL_INPUT => CHARACTERS,`

Terminal input is normally processed in units of one line at a time. A process attempting to read from the terminal as an external file is suspended until a complete line has been typed. At that time, the outstanding read call (and possibly also later calls) is satisfied.

The `LINES` option specifies a line-at-a-time data transfer, which is the default case.

The `CHARACTERS` option means that data transfers character by character, and so a complete line does not have to be entered before the read request can be satisfied. For this option, the `BUFFER_SIZE` must be zero.

When the `CHARACTERS` option is specified, the `ICANON` bit is cleared in the `c_lflag` component of the HP-UX `termio` structure. This bit changes the line discipline for the terminal device. Be aware that the line discipline statue is not maintained on a per file basis. Changing the line discipline for one terminal file does effect other terminal files that are actually associated with the same physical terminal device. See *termio(7)* for additional information.

The `TERMINAL_INPUT` attribute is only applicable to Ada files objects other than `STANDARD_INPUT`. The Ada runtime system uses the default `TERMINAL_INPUT` of `LINES` for the Ada file object `STANDARD_INPUT`. The file name `"/dev/tty"` can be used with the appropriate `FORM` parameter to achieve single character I/O on the same terminal device as `STANDARD_INPUT` if `STANDARD_INPUT` has not been redirected.

### F 8.2.9 The FORM Parameter Attribute - File Structuring

This section describes the structure of Ada files. It also describes how to use the `FORM` parameter to effect the structure of Ada files.



### F 8.2.9.1 The Structure of TEXT\_IO Files

There is no FORM parameter to define the structure of text files. A text file consists of a sequence of bytes containing ASCII character codes.

Table F-8 describes the use of the ASCII characters as Ada terminators in text files. The usage of Ada terminators depends on the file's mode (IN\_FILE or OUT\_FILE) and whether it is associated with a terminal device or a mass-storage file.

**Table F-8. Text File Terminators**

File Type	TEXT_IO Functions	Characters
Mass storage files (IN_FILE)	END_OF_LINE	ASCII.LF Physical end of file
	END_OF_PAGE	ASCII.LF ASCII.FF ASCII.LF Physical end of file Physical end of file
	END_OF_FILE	ASCII.LF Physical end of file Physical end of file
Mass storage files (OUT_FILE)	NEW_LINE	ASCII.LF
	NEW_PAGE	ASCII.LF ASCII.FF ASCII.LF Physical end of file
	CLOSE	ASCII.LF Physical end of file
Terminal device (IN_FILE)	END_OF_LINE	ASCII.LF ASCII.FF ASCII.EOT
	END_OF_PAGE	ASCII.FF ASCII.EOT
	END_OF_FILE	ASCII.EOT
Terminal device (OUT_FILE)	NEW_LINE	ASCII.LF
	NEW_PAGE	ASCII.LF ASCII.FF
	CLOSE	ASCII.LF

See Section F 8.1.3.3 for more information about terminators in text files.

### F 8.2.9.2 The Structure of `DIRECT_IO` and `SEQUENTIAL_IO` Files

This section describes use of the `FORM` parameter for binary (sequential or direct access) files. Two `FORM` attributes, `RECORD_SIZE` and `RECORD_UNIT`, control the structure of binary files.

Such a file can be viewed as a sequence or a set of consecutive `RECORDS`. The structure of a record is

[ `HEADER` ] `OBJECT` [ `UNUSED_PART` ]

A record is composed of up to three items:

1. A `HEADER` consisting of two fields (each of 32 bits)
  - The length of the object in bytes.
  - The length of the descriptor in bytes.
2. An `OBJECT` with the exact binary representation of the Ada object in the executable program, possibly including an object descriptor.
3. An `UNUSED_PART` of variable size to permit full control of the record's size.

The `HEADER` is implemented only if the actual parameter of the instantiation of the I/O package is unconstrained.

The file structure attributes take the form:

`RECORD_SIZE` => *size\_in\_bytes*

`RECORD_UNIT` => *size\_in\_bytes*

The attributes' meaning depends on the object's type (constrained or unconstrained) and the file access mode (sequential or direct access).

There are four types of access that are possible:

- Sequential access of fixed size, constrained objects.
- Sequential access of varying size, unconstrained objects, with objects rounded up to a multiple of the `RECORD_UNIT` size.
- Direct access of fixed size, constrained objects.
- Direct access of fixed size, unconstrained objects, with a maximum size for the object.

The consequences of the above are listed in Table F-9.

**Table F-9. Structuring Binary Files with the FORM Parameter**

Object Type	File Access Mode	RECORD_UNIT Attribute	RECORD_SIZE Attribute
Constrained	Sequential I/O Direct I/O	The RECORD_UNIT attribute is illegal.	<p>If the RECORD_SIZE attribute is omitted, no UNUSED_PART is implemented. The default RECORD_SIZE is the object's size.</p> <p>If present, the RECORD_SIZE attribute must specify a record size greater than or equal to the object's size. Otherwise, the exception USE_ERROR is raised.</p>
Unconstrained	Sequential I/O	<p>By default, the RECORD_UNIT attribute is one byte.</p> <p>The size of the record is the smallest multiple of the specified (or default) RECORD_UNIT that holds the object and its length. This is the only case where different records in a file can have different sizes.</p>	The RECORD_SIZE attribute is illegal.
Unconstrained	Direct I/O	The RECORD_UNIT attribute is illegal.	<p>The RECORD_SIZE attribute has no default value, and if a value is not specified, a USE_ERROR is raised.</p> <p>If you attempt to input or output an object larger than the given RECORD_SIZE, a DATA_ERROR exception is raised.</p>

## F 9. The Ada/800 Development System and HP-UX Signals

The Ada/800 runtime in the HP 9000 Series 800 uses HP-UX signals to implement the following features of the Ada language:

- Ada exception handling
- Ada task management
- Ada delay timing

### F 9.1 HP-UX Signals Reserved by the Ada Runtime

The HP-UX signals reserved and used by the Ada runtime are listed in Table F-10.

Table F-10. Ada/800 Signals

Signal	Description
SIGALRM	Used for <b>delay</b> in tasking programs.
SIGVTALRM	Used for task scheduling (time slicing) in tasking programs.
SIGFPE	Causes the exceptions <code>CONSTRAINT_ERROR</code> , <code>NUMERIC_ERROR</code> , <code>STORAGE_ERROR</code> , or <code>PROGRAM_ERROR</code> .
SIGSEGV	Causes the exception <code>PROGRAM_ERROR</code> .
SIGBUS	Causes the exception <code>PROGRAM_ERROR</code> .
SIGILL	Causes the exception <code>PROGRAM_ERROR</code> .

#### NOTE

The signals SIGSEGV, SIGBUS, and SIGILL are not reserved by the Ada runtime. These signals are never deliberately produced by generated code or by the Ada runtime to cause an exception to be raised. However, if due to a programming error, access is attempted to misaligned or protected data (causing SIGSEGV or SIGBUS) or an illegal instruction is executed (causing SIGILL); `PROGRAM_ERROR` will be raised unless the application has overridden the Ada runtime and specified some other action when receiving these signals.

**NOTE**

The signals listed in Table F-10 will induce an exception even if non-Ada code is executing at the time the signal is received. If interface code causes one of these signals or is running when a signal is received from an outside source, the Ada code that called the interface code will receive an Ada exception.

**NOTE**

The SIGALRM and SIGVTALRM signals are not always generated by a tasking program. SIGALRM is only generated if and when a **delay** statement is encountered in a tasking program. SIGVTALRM is only generated if time slicing was enabled when the program was bound (time slicing is enabled by default and can be disabled with the binder option *-W b,-s,0*.) A sequential (non-tasking) program does not use either the SIGALRM or SIGVTALRM signals and they are not reserved by the Ada runtime in such sequential programs.

## F 9.2 HP-UX Signals Used for Ada/800 Exception Handling

The Ada/800 implementation uses both signals and various procedure calls to raise exceptions. The Ada runtime handlers for the signals that raise exceptions are set during the elaboration of the Ada runtime system. Defining a new handler (or changing the signal action to SIG\_DFL or SIG\_IGN) for any of these signals can subvert the normal exception handling mechanism of Ada and might result in an erroneous runtime execution.

When your Ada code uses external interfaced subprograms, you must ensure that these external interfaced subprograms do *not* redefine the signal handlers (or signal action) for any of the HP-UX signals reserved by the Ada runtime.

The SIGFPE signal has a predefined meaning and is reserved by the Ada runtime for exception handling. The SIGFPE signal is generated in your compiled Ada code whenever one of the predefined runtime checks fails, including null access value checks. The runtime examines the context in which the signal occurred and raises the appropriate one of CONSTRAINT\_ERROR, NUMERIC\_ERROR, STORAGE\_ERROR, or PROGRAM\_ERROR. An unexpected SIGFPE signal that was generated outside of Ada code or sent to the process from an outside source will usually cause the exception PROGRAM\_ERROR to be raised. If the unexpected signal occurred in a context where the runtime believed a legitimate exception could have occurred, that exception might be raised instead of PROGRAM\_ERROR.

The signals SIGSEGV and SIGBUS are generated by access to an illegal or improperly aligned address. Normally these signals will not be generated in an Ada program because access values are initialized to null and are only assigned legal and properly aligned values by generated code, and have runtime checks performed on them to detect attempts to dereference a null access value (causing CONSTRAINT\_ERROR via SIGFPE as mentioned above.) Such illegal or improperly aligned addresses are usually produced by the improper use of UNCHECKED\_CONVERSION or are supplied by interfaced code. In response to receiving SIGSEGV, the Ada runtime will raise PROGRAM\_ERROR. An unexpected SIGSEGV signal that was generated outside of Ada code or sent to the process from an outside source will also cause the exception PROGRAM\_ERROR to be raised.

The signal SIGILL is generated by the execution of an illegal instruction. Normally this signal will not be generated in an Ada program because generated Ada code does not contain any illegal instructions. Execution of an illegal instruction usually occurs in interfaced code. In response to receiving SIGILL, the Ada runtime will raise PROGRAM\_ERROR. An unexpected SIGILL signal that was generated outside of Ada code or sent to the process from an outside source will also cause the exception PROGRAM\_ERROR to be raised.

### NOTE

User code can define its own handler (or change the signal action) for SIGSEGV, SIGBUS, and SIGILL without directly compromising the operation of the Ada program. However, ignoring a synchronous instance of one of these signals or continuing execution after handling a synchronous instance of one of these signals is not advised without a thorough understanding of the causes and continuation strategies for such signals under HP-UX on HP-PA.

NOTE
------

The Ada/800 binder does not specify `-z` or `-Z` to the linker (`ld(1)`) to control the system action on a dereference of a null pointer. Either the `ld(1)` default or a user-specified value (via `-W 1`) will therefore take effect.

If Ada code is compiled with checks enabled (the default case), the Ada/800 runtime system will operate properly with either `-z` or `-Z` behavior specified to the linker because Ada will generate software checks for null pointer dereferencing.

If Ada code is compiled with pointer dereference checks disabled (using the `-C` or `-R` compiler options or using pragma `SUPPRESS`), some null pointer checking can be restored with essentially no runtime overhead by using the `-z` linker option.

If `-z` is specified, the system will send `SIGSEGV` if a null pointer is dereferenced; the Ada/800 runtime system will map that signal to `PROGRAM_ERROR`. However, the Ada software checks for null pointer dereferencing are intended to handle all cases where a null pointer could appear and will raise `CONSTRAINT_ERROR` if such a dereference occurs. The `SIGSEGV` enabled by the `-z` linker option will only be sent when the final result of an address calculation is the null pointer and the resulting exception will be `PROGRAM_ERROR` instead of `CONSTRAINT_ERROR`.

### F 9.3 HP-UX Signals Used for Ada Task Management

The HP-UX virtual alarm facility is used by the Ada runtime for task management. When the Ada program contains tasks, the HP-UX signal `SIGVTALRM` is used to implement time slicing. Under a time slicing algorithm, the Ada runtime allocates the available processor time among concurrent tasks. If the Ada program does not contain tasks, it is a sequential program.

For the case of a sequential program or a tasking program with time slicing *disabled*, the HP-UX signal `SIGVTALRM` is *not* reserved by the Ada runtime. For the case of a tasking program with time slicing *enabled*, the HP-UX signal `SIGVTALRM` is reserved by the Ada runtime.

When your Ada code uses external interfaced subprograms, you must ensure that these external interfaced subprograms *do not* redefine the signal handlers for any of the HP-UX signals reserved by the Ada runtime. If you redefine a handler for `SIGVTALRM` and your Ada program is using task time slicing, unpredictable program behavior will result.

#### **F 9.4 HP-UX Signals Used for Ada Delay Timing**

The HP-UX signal SIGALRM is used by the Ada runtime to time **delay** statements in your Ada source code in a tasking program. The maximum resolution of the timer is 1/100 of a second. Thus, all **delay** statements are implemented using actual delays that are integral multiples of 1/100 of a second. Non-zero delays for periods smaller than 1/100 of a second will delay for at least 1/100 of a second.

The signal SIGALRM is reserved by the Ada runtime if your tasking program contains any **delay** statements. For a sequential program or a tasking program that contains no **delay** statements, the signal SIGALRM is *not* reserved.

When your Ada code uses external interfaced subprograms you must ensure that these external interfaced subprograms *do not* redefine the signal handlers for any of the HP-UX signals reserved by the Ada runtime. If you redefine a handler for SIGALRM and your tasking Ada program is using **delay** statements, unpredictable program behavior will result.

#### **F 9.5 Protecting Interfaced Code from Ada's Asynchronous Signals**

The two signals mentioned above ( SIGALRM and SIGVTALRM ) occur asynchronously. Because of this, they may occur while your code is executing an external interfaced subprogram. For details on protecting your external interfaced subprogram from adverse effects caused by these signals, see the section in the *HP Ada User's Guide* on "Interfaced Subprograms and Ada's Use of Signals."

#### **F 9.6 Programming in Ada With HP-UX Signals**

If you intend to utilize signals in external interfaced subprograms, refer to Section F 11.7, "Potential Problems Using Interfaced Subprograms." This version of the product does not support the association of an HP-UX signal such as SIGINT with an Ada procedure or a task entry. The signal must be handled inside an external subprogram. The same cautions apply for this external subprogram as for any external interfaced subprogram that might be interrupted by an unexpected signal.



## F 10. Limitations

This section lists limitations of the compiler and the Ada development environment.

### F 10.1 Compiler Limitations

NOTE
------

It is impossible to give exact numbers for most of the limits listed in this section. The various language features may interact in complex ways to lower the limits.

The numbers represent "hard" limits in simple program fragments devoid of other Ada features.

Limit	Description
255	Maximum number of characters in a source line.
253	Maximum number of characters in a string literal.
255	Maximum number of characters in an enumeration type element.
32767	In an enumeration type, the sum of the lengths of the IMAGE attributes of all elements in the type, plus the number of elements in the type, must not exceed this value.
2047	Maximum number of actual compilation units in a library.
32767	Maximum number of enumeration elements in a single enumeration type (this limit is further constrained by the maximum number of characters for all enumeration literals of the type).
2047	Maximum number of "created" units in a single compilation.
2**31-1	Maximum number of bits in any size computation.
2048	Links in a library.
2048	Libraries in the INSTALLATION family (250 of which are reserved).
2047	Libraries in either the PUBLIC or a user defined family. (For more information, see the <i>Ada User's Guide</i> , which discusses families of Ada libraries and the supported utilities (tools) to manage them).

## Implementation-Dependent Characteristics

Limit	Description
	- Maximum number of tasks is limited only by heap size.
255	Maximum number of characters in any path component of a file specified for access by the Ada compiler. If a component exceeds 255 characters, NAME_ERROR will be raised.
1023	The maximum number of characters in the entire path to a file specified for access by the Ada compiler. If the size of the entire path exceeds 1023 characters, NAME_ERROR will be raised.
	The pathname limits apply to the entire path during and after the resolution of symbolic links and context-dependent files (CDFs) if they appear in the specified path.

The following items are limited only by overflow of internal tables (AIL or HLST tables). All internal data structures of the compiler which previously placed fixed limits are now dynamically created.

- Maximum number of identifiers in a unit. An identifier includes enumerated type identifiers, record field definitions, and (generic) unit parameter definitions.
- Maximum "structure" depth. Structure includes the following: nested blocks, compound statements, aggregate associations, parameter associations, subexpressions.
- Maximum array dimensions. Set to maximum structure depth/10.\*
- Maximum number of discriminants in a record constraint.\*
- Maximum number of associations in a record aggregate.\*
- Maximum number of parameters in a subprogram definition.\*
- Maximum expression depth.\*
- Maximum number of nested frames. Library-level unit counts as a frame.
- Maximum number of overloads per compilation unit.
- Maximum number of overloads per identifier.

\* A limit on the size of tables used in overloading resolution can potentially lower this figure. This limit is set at 500. It reflects the number of possible interpretations of names in any single construct under analysis by the compiler (procedure call, assignment statement, and so on.)

## F 10.2 Ada Development Environment Limitations

The following limits apply to the Ada development environment (*ada.umgr(1)*, *ada.fmgr(1)*, Ada tools).

Limit	Description
200	The number of characters in the actual rooted path of an Ada program LIBRARY or FAMILY of libraries.
200	The number of characters in the string (possibly after expansion by an HP-UX shell) specifying the name of an Ada program LIBRARY or FAMILY of libraries. This limit applies to strings (pathname expressions) specified for a LIBRARY or FAMILY that you submit to tools such as <i>ada.mklib(1)</i> or <i>ada.umgr(1)</i> .
512	Maximum length of an input line for the tools <i>ada.fmgr(1)</i> and <i>ada.umgr(1)</i> .
255	The maximum number of characters in any path component of a file specified for access by an Ada development environment tool. If a component exceeds 255 characters, NAME_ERROR will be raised.
1023	The maximum number of characters in the entire path to a file specified for access by an Ada program or an Ada development environment tool. If the size of the entire path exceeds 1023 characters, NAME_ERROR will be raised.

The pathname limits apply to the entire path during and after the resolution of symbolic links and context-dependent files (CDFs) if they appear in the specified path.

### **F 10.3 Limitations Affecting User-Written Ada Applications**

The Ada/800 compiler and Ada development environment is expected to be used on versions of the HP-UX operating system that support Network File Systems (NFS), diskless HP-UX workstations, long filename file systems and symbolic links to files. To accomodate this diversity within a file system used in both the development and target systems, the HP Ada compiler places some restrictions on the use of the OPEN and CREATE on external files. This section describes those restrictions.

#### **F 10.3.1 Restrictions Affecting Opening or Creating Files**

Unless you observe the following restrictions on the size of path components and file names, the OPEN or CREATE call will raise NAME\_ERROR in certain situations.

##### **F 10.3.1.1 Restrictions on Path and Component Sizes**

The maximum number of characters in any path component of a file specified for access by an Ada program is 255.

The maximum number of characters in the entire path to a file specified for access by an Ada program is 1023.

The pathname limits apply to the entire path during and after the resolution of symbolic links and context-dependent files (CDFs) if they appear in the specified path.

##### **F 10.3.1.2 Conditions that Raise NAME\_ERROR**

When using OPEN and CREATE, the Ada exception NAME\_ERROR will be raised if any path component exceeds 255 characters or if the entire path exceeds 1023 characters.

When opening a file, the Ada exception NAME\_ERROR will be raised if there are any directories in the rooted path of the file that are not readable by the "effective uid" of the program. This restriction applies to intermediate path components that are encountered during the resolution of symbolic links.

##### **F 10.3.2 Restrictions on TEXT\_IO.FORM**

The function TEXT\_IO.FORM will raise USE\_ERROR if it is called with either of the predefined files, STANDARD\_INPUT or STANDARD\_OUTPUT.

### **F 10.3.3 Restrictions on the Small of a Fixed Point Type**

A length clause may be used to specify the value to use for 'SMALL on a fixed point type. However, this implementation requires that the value specified for 'SMALL is a power of two. The compiler rejects a compilation unit with a length clause specification with an IMPLEMENTATION RESTRICTION if 'SMALL is not an exact power of two.

### **F 10.3.4 Record Type Change of Representation**

In the current version of the compiler, it is not possible to apply a record representation clause to a derived record type. The compiler will use the same storage representation for all records of the same base type. Thus, the compiler does not support the "Change in Representation" as described in the *Ada RM*, Section 13.6.

### **F 10.3.5 Record Type Alignment Clause**

A record type alignment clause can specify that a record type is byte, half-word, word, or double-word aligned (specified as 1, 2, 4, or 8 bytes). *Ada/800* does not support alignments larger than a 8-byte alignment.

### **F 10.3.6 Pragma INTERFACE on Library Level Subprograms**

In the current version of the compiler, it is not possible to supply a pragma INTERFACE to a library-level subprogram. Any subprogram that a pragma INTERFACE is applied to must be contained within an Ada compilation unit, usually a package.

## F 11. Calling External Subprograms From Ada

In general, in Ada/800, parameters of external interfaced subprograms are passed according to the standard HP-PA calling conventions (see *HP-PA Procedure Calling Convention Reference Manual*). This convention is used by Hewlett-Packard for other language products on the HP 9000 Series 800 family of computers. The languages described in this section are the HP implementations of HP-PA Assembler, HP C, HP FORTRAN 77, and HP Pascal on the HP-UX Series 800 systems.

Ada parameter passing deviates from the standard HP-PA calling conventions when passing arrays and records that occupy 64 bits or less. Ada passes all arrays and records by reference; therefore, arrays and records that occupy 64 bits or less are not passed by copy as per the standard HP-PA calling convention. Such arrays and records cannot be passed to interfaced subprograms that expect objects of these types to be passed by copy (for example an HP C or HP Pascal subprogram that declares such parameter to be passed by value). Such arrays and records can be passed to interfaced subprograms that expect a reference to such an object to be passed.

When you specify the interfaced language name, that name is used to select the correct calling conventions for supported languages. Subprograms written in HP-PA Assembler, HP C, HP FORTRAN 77, and HP Pascal interface correctly with the Ada/800 subprogram caller. This section contains detailed information about calling subprograms written in these languages. If the subprogram is written in a language from another vendor, you must follow the standard calling conventions.

In the Ada/800 implementation of external interfaced subprograms, the three Ada parameter passing modes (*in*, *out*, *in out*) are supported, with some limitations as noted below. Scalar and access parameters of mode *in* are passed by *value*. All other parameters of mode *in* are passed by *reference*. Parameters of mode *out* or *in out* are always passed by reference. (See Table F-11 and Figure F-1 for details.)

**Table F-11. Ada Types and Parameter Passing Modes**

Ada Type	Mode Passed By Value	Mode Passed By Reference
SCALAR, ACCESS	in	out, in out
All others except TASK and FIXED POINT		in, out, in out
TASK and FIXED POINT	(not passed)	(not passed)

The values of the following types *cannot* be passed as parameters to an external interfaced subprogram:

- Task types (*Ada RM*, Section 9.1 and 9.2),
- Fixed point types (*Ada RM*, Section 3.5.9 and 3.5.10).

A composite type (an array or record type) is always passed by reference (as noted above). A component of a composite type is passed according to its type classification (scalar, access, or composite).

Only scalar types (enumeration, character, Boolean, integer, or floating point) or access types are allowed for the result returned by an external function subprogram.

**NOTE**

There are no checks for consistency between the subprogram parameters (as declared in Ada) and the corresponding external subprogram parameters. Because external subprograms have no notion of Ada's parameter modes, parameters passed by reference are not protected from modification by an external subprogram. Even if the parameter is declared to be only of mode `in` (and not `out` or `in out`) but is passed by reference (that is, an array or record type), the value of the Ada actual parameter can still be modified.

The possibility that the parameter's actual value will be modified by an external interfaced subprogram exists when that parameter is not passed by value. Objects whose attribute `'ADDRESS` is passed as a parameter and parameters passed by reference are not protected from alteration and are subject to modification by the external subprogram. In addition, such objects will have no run-time checks performed on their values upon return from interfaced external subprograms.

Erroneous results may occur if the parameter values are altered in some way that violates Ada constraints for the actual Ada parameter. The responsibility is yours to ensure that values are not modified in external interfaced subprograms in such a manner as to subvert the strong typing and range checking enforced by the Ada language.

**CAUTION**

Be very careful to establish the exact nature of the types of parameters to be passed. The bit representations of these types can be different between this implementation of Ada and other languages, or between different implementations of the Ada language. Pay careful attention to the size of parameters because parameters must occupy equal space in the interfaced language. When passing record types, pay particular attention to the internal organization of the elements of a record because Ada semantics do not guarantee a particular order of components. Moreover, Ada compilers are free to rearrange or add components within a record. See Section F 4, "Type Representation", for more information.

## F 11.1 General Considerations in Passing Ada Types

Section F 11.1 discusses each data type in general terms. Sections F 11.2 through F 11.5 describe the details of interfacing your Ada programs with external subprograms written in HP-PA Assembler, HP C, HP FORTRAN 77, and HP Pascal. Section F 11.6 provides summary tables.

The Ada types are described in the following order:

- Scalar
  - Integer
  - Enumeration
  - Boolean
  - Character
  - Real
- Access
- Array
- Record
- Task

### F 11.1.1 Scalar Types

This section describes general considerations when you are passing scalar types between Ada programs and subprograms written in a different HP language. The class scalar types includes integer, real, and enumeration types. Because character and Boolean types are predefined Ada enumeration types, they are also scalar types.

Scalar type parameters of mode `in` are passed by value. Scalar type parameters of mode `in out` or `out` are passed by reference.

#### F 11.1.1.1 Integer Types

In Ada/800, all integers are represented in two's complement form. The type `SHORT_SHORT_INTEGER` is represented as an 8-bit quantity, the type `SHORT_INTEGER` is represented as a 16-bit quantity, and the type `INTEGER` is represented as a 32-bit quantity.

All integer types can be passed to interfaced subprograms. When an integer is used as a parameter for an interfaced subprogram, the call can be made either by reference or by value. If passed by reference, the value of the actual integer parameter is not copied or modified, but a 32-bit address pointer to the integer value is passed. If passed by value, a copy of the actual integer parameter value is passed, based on its size, as per the standard HP-PA calling convention. If passed in a register, it will be sign extended as required. See Sections F 11.2.1.1, F 11.3.1.1, F 11.4.1.1, and F 11.5.1.1 for details specific to interfaced subprograms written in different languages.

Integer types may be returned as function results from external interfaced subprograms.



### F 11.1.1.2 Enumeration Types

Values of an enumeration type (*Ada RM*, Section 3.5.1) without an enumeration representation clause (*Ada RM*, Section 13.3) have an internal representation of the value's position in the list of enumeration literals defining the type. These values are non-negative. The first literal in the list corresponds to an integer value of zero.

An enumeration representation clause can be used to further control the mapping of internal codes for an enumeration identifier. See Section F 4.1, "Enumeration Types," for information on enumeration representation clauses.

Values of enumeration types are represented internally as either an 8-, 16-, or 32-bit quantity (see Section F 4.1, "Enumeration Types"). When an enumeration value is used as a parameter for an interfaced subprogram, the call can be made either by reference or by value. If passed by reference, the value of the actual enumeration parameter is not copied or modified, but a 32-bit address pointer to the enumeration value is passed. If passed by value, a copy of the actual enumeration parameter value is passed, based on its size, as per the standard HP-PA calling convention. If passed in a register, it will be zero extended as required. See Sections F 11.2.1.1, F 11.3.1.1, F 11.4.1.1, and F 11.5.1.1 for details specific to interfaced subprograms written in different languages.

Enumeration types may be returned as function results from external interfaced subprograms.

### F 11.1.1.3 Boolean Types

Values of the predefined enumeration type `BOOLEAN` are represented internally as an 8-bit quantity. The Boolean value `FALSE` is represented by the 8-bit value `2#0000_0000#` and the Boolean value `TRUE` is represented by the 8-bit value `2#0000_0001#`. This representation is the same as that of any two-valued enumeration type whose size and internal code values have not been modified with a representation clause.

Boolean values are passed the same as any other enumeration values.

Boolean types can be returned as function results from external interfaced subprograms.

### F 11.1.1.4 Character Types

The values of the predefined enumeration type `CHARACTER` are represented as 8-bit values in a range 0 through 127.

Values of the character type are passed as parameters and returned as function results as are values of any other 8-bit enumeration type.

Character types may be returned as function results from external interfaced subprograms.

See Sections F 11.2.1.1, F 11.3.1.1, F 11.4.1.1, and F 11.5.1.1 for details specific to interfaced subprograms written in different languages.

### F 11.1.1.5 Real Types

Ada fixed point types and Ada floating point types are discussed in the following subsections.

#### Fixed Point Types

Ada fixed point types (*Ada RM*, Section 3.5.9 and 3.5.10) are not supported as parameters or as results of external interfaced subprograms.

Fixed point types *cannot* be returned as function results from external interfaced subprograms.

#### Floating Point Types

Floating point values (*Ada RM*, Sections 3.5.7 and 3.5.8) in the HP implementation of Ada are of 32 bits (FLOAT) or 64 bits (LONG\_FLOAT). These two types conform to the *IEEE Standard for Binary Floating-Point Arithmetic*.

The Ada type FLOAT is a 32-bit real type and is passed as a 32-bit real; this type is *never* extended to a 64-bit real. The Ada type LONG\_FLOAT is a 64-bit real type and is passed as a 64-bit real.

Both floating point types can be passed to interfaced subprograms. When a floating point value is used as a parameter for an interfaced subprogram, the call can be made either by reference or by value. If passed by reference, the value of the actual floating point parameter is not copied or modified; a 32-bit address pointer to the floating point value is passed. If passed by value, a copy of the actual floating point parameter value is passed, based on its size, as per the standard HP-PA calling convention.

See Sections F 11.2.1.5, F 11.3.1.5, F 11.4.1.5, and F 11.5.1.5 for details specific to interfaced subprograms written in different languages.

Floating point types may be returned as function results from external interfaced subprograms, with some restrictions. See Section F 11.3.1.5, "Real Types and HP C Subprograms," for details.

### F 11.1.2 Access Types

Values of an access type (*Ada RM*, Section 3.8) have an internal representation which is the 32-bit address of the underlying designated object.

An object's address can be retrieved by applying the `'ADDRESS` attribute to the object. In the case of an access type object, you may want either the address of the access type object or the address of the underlying object that it points to. The underlying object's address can be retrieved by applying the attribute `'ADDRESS` in this way:

```
access_object.all'ADDRESS.
```

The use of `.all` implies that the `'ADDRESS` operation applies to the contents of the access type object and not to the access type object itself.

An access type object has a value that is the address of the designated object. Therefore, when an access type is passed by value, a copy of this 32-bit address is passed. If an access type object is passed by reference, however, the address of the access type object itself is passed. This will effectively force references to the designated object to be double indirect references. See Figure F-1 for details.

**Figure F-1. Passing Access Types to Interfaced Subprograms**

Access types may be returned as function results from external interfaced subprograms.

Ada access types are pointers to Ada objects. In the implementation of HP Ada for the Series 800 Computer System, an address pointer value will always point at the first byte of storage for the designated object and not at a descriptor for the object. This may not be the case for other implementations of the Ada language and should be considered when Ada source code portability is an issue.

### NOTE

If a pointer to an unconstrained array object is passed to interfaced code, the information that describes the runtime constraints needs to be passed explicitly.

## F 11.1.3 Array Types

In the HP implementation of Ada, arrays (*Ada RM*, Section 3.6) are always passed by reference. The value passed is the address of the first element of the array. When an array is passed as a parameter to an external interfaced subprogram, the usual checks on the consistency of array bounds between the calling program and the called subprogram are not enforced. You are responsible for ensuring that the external interfaced subprogram keeps within the proper array bounds. You may need to explicitly pass the upper and lower bounds for the array type to the external subprogram.

The external subprogram should access and modify such an array in a manner appropriate to the actual Ada type. Note that Ada will *not* range check the values that may have been stored in the array by the external subprogram. In Ada range checks are only required when assigning an object with a constraint, thus range checks are not performed when reading the value of an object with a constraint. If an external subprogram modifies elements in an Ada array object, it has the responsibility to ensure that any values stored meet the type constraints imposed by the Ada type.

Array element allocation, layout, and alignment are described in Section F 4.7, "Array Types."

Values of the predefined type *STRING* (*Ada RM*, Section 3.6.3) are unconstrained arrays and are passed by reference as described above. The address of the first character in the string is passed. You may need to explicitly pass the upper and lower bounds, or the length of the string to the external subprogram.

Returning strings from an external interfaced subprogram to Ada (such as *OUT* parameters) is not supported. See Section F 11.3.3 for a complete example which shows how to return *STRING* type information from interfaced subprograms.

Array types *cannot* be returned as function results from external interfaced subprograms. However, an access type to the array type can be returned as a function result.

## F 11.1.4 Record Types

Records (*Ada RM*, Section 3.7) are always passed by reference in the HP implementation of Ada, passing the 32-bit address of the first component of the record.

The external subprogram should access and modify such a record in a manner appropriate to the actual Ada type. Note that Ada will *not* range check the values that may have been stored in the record by the external subprogram. In Ada, range checks are only required when assigning an object with a constraint,

thus range checks are not performed when reading the value of an object with a constraint. If an external subprogram modifies a component in an Ada record object, it has the responsibility to ensure that any values stored meet the type constraints imposed by the Ada type for that component.

When interfacing with external subprograms using record types, it is recommended that you provide a complete record representation clause for the record type. It is also your responsibility to ensure that the external subprogram accesses the record type in a manner that is consistent with the record representation clause. For a complete description of record representation clauses see Section F 4. 8, "Record Types."

If a record representation clause is *not* used, you should be aware that the individual components of a record may have been reordered internally by the Ada compiler. This means that the implementation of the record type may have components in an different order than the declarative order. Ada semantics do not require a specific ordering of record components.

When interfacing record types with external subprograms, you may want to communicate some or all of the offsets of individual record components. One reason for doing this would be to avoid duplicating the record information in two places: once in your Ada code and again in the interfaced code. Software maintenance is often complicated by this practice.

The attribute 'POSITION returns the offset of a record component with respect to the starting address of the record. By passing this information to the external subprogram, you can avoid duplicating the record type definition in your external subprogram.

The starting address of a record type can be passed to an external subprogram in one of three ways:

- the record object passed as a parameter (records are always passed by reference).
- the attribute 'ADDRESS of the record object passed as a parameter.
- a *value* parameter that is of an access type to the record object.

Direct assignment to a discriminant of a record is not allowed in Ada (*Ada RM*, Section 3.7.1). A discriminant *cannot* be passed as an actual parameter of mode *out* or *in out*. This restriction applies equally to Ada/800 subprograms and to external interfaced subprograms written in other languages. If an interfaced program is given access to the whole record (rather than individual components), that code should *not* change the discriminant value, because that would violate the Ada standard rules for discriminant records.

In Ada/800, records are packed and variant record parts are overlaid; the size of the record is the longest variant part. If a record contains discriminants or composite components having a dynamic size, the compiler may add implicit components to the record. See Section F 4. 8, "Record Types," for a complete discussion of these components.

Dynamic components and components whose size depends upon record discriminant values are implemented indirectly within the record by using implicit 'OFFSET components.

Record types *cannot* be returned as function results from external interfaced subprograms. However, an access type to the record type can be returned as a function result.

### F 11.1.5 Task Types

A task type *cannot* be passed to an external procedure or external function as a parameter in Ada/800. A task type *cannot* be returned as a function result from an external function.

## **F 11.2 Calling Assembly Language Subprograms**

When calling interfaced assembly language subprograms, specify the named external subprogram in a compiler directive:

```
pragma INTERFACE ( ASSEMBLER, Ada_subprogram_name );
```

Note that the language type specification is ASSEMBLER and not ASSEMBLY. This description refers to the HP assembly language for the HP-PA processor family upon which the Series 800 family is based.

Interfaced subprograms written in HP-PA Assembly Language that conform to the HP-PA procedure calling conventions can be called from Ada with no special precautions. See the *HP-PA Procedure Calling Convention Reference Manual* and the *HP-PA Assembly Language Reference Manual* for additional information.

Only scalar types (integer, floating point, character, Boolean, and enumeration types) and access types are allowed as result types for an external interfaced function subprogram written in HP-PA Assembly Language.

### **F 11.2.1 Scalar Types and Assembly Language Subprograms**

See Section F 11.1 for details.

#### **F 11.2.1.1 Integer Types and Assembly Language Subprograms**

See Section F 11.1.1 for details.

#### **F 11.2.1.2 Enumeration Types and Assembly Language Subprograms**

See Section F 11.1.1.2 for details.

**F 11.2.1.3 Boolean Types and Assembly Language Subprograms**

See Section F 11.1.1.3 for details.

**F 11.2.1.4 Character Types and Assembly Language Subprograms**

See Section F 11.1.1.4 for details.

**F 11.2.1.5 Real Types and Assembly Language Subprograms**

See Section F 11.1.1.5 for details.

**F 11.2.2 Access Types and Assembly Language Subprograms**

See Section F 11.1.2 for details.

**F 11.2.3 Array Types and Assembly Language Subprograms**

See Section F 11.1.3 for details.

**F 11.2.4 Record Types and Assembly Language Subprograms**

See Section F 11.1.4 for details.

### F 11.3 Calling HP C Subprograms

When calling interfaced HP C subprograms, the form

```
pragma INTERFACE (C, Ada_subprogram_name)
```

is used to identify the need to use the HP C parameter passing conventions.

To call the following HP C subroutine

```
void c_sub (val_parm, ref_parm)
int val_parm;
int *ref_parm;
{
    ...
}
```

Ada requires an interfaced subprogram declaration:

```
procedure C_SUB (VAL_PARAM : in INTEGER;
                 REF_PARAM : in out INTEGER);
pragma INTERFACE (C, C_SUB);
```

In the above example we provided the Ada subprogram identifier `C_SUB` to the `pragma INTERFACE`. If a `pragma INTERFACE_NAME` is not supplied, the HP C subprogram name is the name of the Ada subprogram specified in the `pragma INTERFACE`, with all alphabetic characters shifted to lowercase.

Note that the parameter in the preceding example, `VAL_PARAM`, must be of mode `in`, to match the parameter definition for `val_parm` found in the HP C subroutine. Likewise, `REF_PARAM`, must be of mode `in out` to correctly match the C definition of `*ref_parm`. Also, note that the names for parameters *do not* need to match exactly. However, the mode of access and the data type *must* be correctly matched, but there is no compile-time or run-time check that can ensure that they match. It is your responsibility to ensure their correctness.

You must use `pragma INTERFACE_NAME` whenever the HP C subprogram name contains characters not acceptable within Ada identifiers or when the HP C subprogram name contains uppercase letter(s). You can also use a `pragma INTERFACE_NAME` if you want your Ada subprogram name to be different than the HP C subprogram name.

Note that the Ada/800 compiler does *not* automatically convert 32-bit real parameters to 64-bit real parameters. See Section F 11.3.1.5, "Real Types and HP C Subprograms," for details.

Only scalar types (integer, floating point, character, Boolean, and enumeration types) and access types are allowed as result types for an external interfaced function subprogram written in HP C.

When binding and linking Ada programs with interfaced subprograms written in HP C, the libraries `libc.a`, `libm.a`, and `libcl.a` are usually required. The Ada/800 binder will automatically provide the `-lm -lc -lcl` directives to the linker. You are not required to specify `"-lm -lc -lcl"` when binding and linking the Ada program on the `ada(1)` command line.



For more information about C language interfacing, see the following manuals: *HP-UX Concepts and Tutorials: Programming Environment*, *HP-UX Concepts and Tutorials: Device I/O and User Interfacing*, and *HP-UX Portability Guide*. For general information about passing Ada types, see Section F 11.1.

### F 11.3.1 Scalar Types and HP C Subprograms

See Section F 11.1.1 for details.

#### F 11.3.1.1 Integer Types and HP C Subprograms

See Section F 11.1.1.1 for details.

When passing integers by reference, note that an Ada `SHORT_SHORT_INTEGER` (eight bits) actually corresponds with the HP C type `char`, because C treats this type as a numeric type.

Table F-12 summarizes the integer correspondence between Ada and C.

**Table F-12. Ada/800 versus HP C Integer Correspondence**

Ada	HP C	Bit Length
CHARACTER	<code>char</code>	8
SHORT_SHORT_INTEGER	<code>char</code>	8
SHORT_INTEGER	<code>short</code> and <code>short int</code>	16
INTEGER	<code>int</code> , <code>long</code> , and <code>long int</code>	32

All Ada integer types are allowed for the result returned by an external interfaced subprogram written in HP C if care is taken with respect to differences in the interpretation of 8-bit quantities.

#### F 11.3.1.2 Enumeration Types and HP C Subprograms

See Section F 11.1.1.2 for details.

HP C enumeration types have the same representation as Ada enumeration types. They both are represented as unsigned integers beginning at zero. In HP C, the size of an enumeration type is always 32 bits. When HP C passes enumeration types as *value* parameters, the values are zero extended to 32 bits. Because Ada also performs the zero extension to 32 bits for enumeration type values, they will be in the correct form for HP C subprograms. If a representation specification applies to the Ada enumeration type, the value specified by the representation clause (not the `'POS` value) will be passed to the HP C routine.

### F 11.3.1.3 Boolean Types and HP C Subprograms

See Section F 11.1.1.3 for details.

Booleans are passed as other enumeration types are passed; see Section F 11.3.1.2 for details.

The type Boolean is not defined in HP C and the Ada/800 representation of Booleans does not directly correspond to any type in HP C. However, an Ada Boolean could be represented in C with an appropriate two-valued enumeration type or with an HP C integer type.

Boolean types are allowed for the result returned by an external interfaced subprogram written in HP C, when care is taken to observe the internal representation.

### F 11.3.1.4 Character Types and HP C Subprograms

See Section F 11.1.1.4 for details.

The Ada predefined type CHARACTER and any of its subtypes correspond with the type char in HP C. Both the Ada and HP C types have the same internal representation and size. However, in Ada the type CHARACTER is constrained to be within the 128 character ASCII standard.

### F 11.3.1.5 Real Types and HP C Subprograms

This section discusses passing fixed point types and floating point types to HP C.

#### Fixed Point Types

Ada fixed point types are not supported as parameters or as results of external subprograms. Ada fixed point types *cannot* be returned as function results from interfaced subprograms written in HP C.

#### Floating Point Types

See Section F 11.1.1.5 for details.

When HP C is operating in compatibility mode (non-ANSI mode), the default calling convention for passing parameters of floating point types by *value* requires that 32-bit single precision reals be converted to 64-bit double precision reals before being passed. Additionally, the convention for functions returning values of real types requires that a 64-bit double precision real be returned and converted to 32 bits if the function was to return a 32-bit real value.

When HP C is operating in ANSI conformant mode (or in compatibility mode with the +r flag specified), 32-bit single precision reals are passed as parameters and are returned without being converted to or from 64-bit double precision.

Consequently, an interface parameter of type FLOAT or of a type derived from a type whose base type is FLOAT, can only be passed directly to float parameters of HP C code compiled in ANSI conformant mode or in compatibility mode with +r specified. An HP C interface subprogram that wants to return a value of type FLOAT or of a type derived from a type whose base type is FLOAT to Ada, can only do so if the HP C code was compiled in ANSI conformant mode or in compatibility mode with +r specified.

To interface with default compatibility mode HP C code (no `+r` specified), the type `LONG_FLOAT` must be used for all parameters of HP C type `float` and for all interface function results of HP C type `float`.

This limitation on passing the Ada type `FLOAT` only applies to parameters that are of the type `FLOAT` or derived from a type whose base type is `FLOAT`. A composite type, such as an array or a record, *can* have components that are of the type `FLOAT`. Also, the type `FLOAT` can be passed by reference to an external HP C subprogram. The HP C calling convention does not require conversion in these cases in any compiler mode.

### F 11.3.2 Access Types and HP C Subprograms

See Section F 11.1.2 for details.

### F 11.3.3 Array Types and HP C Subprograms

See Section F 11.1.3 for details.

Note that constrained Ada arrays with `SHORT_SHORT_INTEGER` or with 8-bit enumeration type components can be most conveniently be associated with an HP C type of the form `char[]` or `char *`.

In Ada/800, the predefined type `STRING` is an unconstrained array type. It is represented in memory as a sequence of consecutive characters without any gaps in between the characters. In HP C, the string type is represented as a sequence of characters that is terminated with an ASCII null character (`\000`). You will need to append a null character to the end of an Ada string if that string is to be sent to an external interfaced HP C subprogram. When retrieving the value of an HP C string object for use as an Ada string, you will need to dynamically allocate a copy of the HP C string. The HP C type `char *` is not compatible with the unconstrained array type `STRING` that is used by Ada.

The examples on the following pages illustrate the handling of strings in HP C and in Ada/800. In the first example, an Ada string is passed to HP C. Note the need to explicitly add a null character to the end of the string so that string will be in the form that HP C expects for character strings.

The HP C routine:

```
/* Receiving an Ada string that has an ASCII.NULL appended to it
   in this C routine
*/

void receive_ada_str (var_str)
    char *var_str;
{
    printf ("C: Received value was : %s \n", var_str);
}
```

## Implementation-Dependent Characteristics

### The Ada routine:

```
-- passing an Ada string to a C routine

procedure SEND_ADA_STR is

    -- Declare an interfaced procedure that sends an
    -- Ada-String to a C-subprogram

    procedure RECEIVE_ADA_STR ( VAR_STR : STRING);
    pragma INTERFACE (C, RECEIVE_ADA_STR);

begin -- SEND_ADA_STR

    -- Test the passing of an Ada string to a C routine
    RECEIVE_ADA_STR ( "Ada test string sent to C " & ASCII.NUL);

end SEND_ADA_STR;
```

In the second example, a C string is converted to an Ada string. Note that Ada must compute the length of the C string and then it must dynamically allocate a new copy of the C string.

### The HP C routine:

```
/* Sending a C string value back to an Ada program */

char *send_c_str()
{
    char *local_string;

    local_string = "a C string for Ada.";
    return local_string;
}
```

The Ada routine:

```

-----
-- We import several useful functions from the package SYSTEM
--   the generic function FETCH
--       to read a character value given an address
--   the function "+"(address, integer)
--       to allow us to index consecutive addresses
--
--   ( See section F 3.1, for the complete specification
--     of the package SYSTEM )
-----

with SYSTEM;
with TEXT_IO;
procedure READ_C_STRING is

    type C_STRING is access CHARACTER; -- This is the C type char *

    type A_STRING is access STRING; -- The Ada type pointer to STRING

    -- Declare an interfaced procedure that returns a pointer
    -- to a C string (actually a pointer to a character)
    function SEND_C_STR return C_STRING;
    pragma INTERFACE (C, SEND_C_STR);

    function FETCH_CHAR is
        new SYSTEM.FETCH (ELEMENT_TYPE => CHARACTER);
        -- Create a non-generic instantiation of the function FETCH

    function C_STRING_LENGTH (SRC : C_STRING) return NATURAL is
        use SYSTEM; -- import the "+"(address, integer) operator
        LEN : NATURAL := 0;
        START : SYSTEM.ADDRESS;
        CUR : CHARACTER;
    begin
        START := SRC.all'ADDRESS;
        loop
            CUR := FETCH_CHAR (FROM => START + INTEGER (LEN));
            exit when CUR = ASCII.NUL;
            LEN := LEN + 1;
        end loop;
        return LEN;
    end C_STRING_LENGTH;
end READ_C_STRING;

```

## Implementation-Dependent Characteristics

```
function CONVERT_TO_ADA (SRC : C_STRING) return A_STRING is
  use SYSTEM; -- import the "+"(address, integer) operator
  A_STORAGE : A_STRING;
  LEN       : NATURAL;
  C_START   : SYSTEM.ADDRESS;
  C_CUR     : CHARACTER;
begin
  LEN := C_STRING_LENGTH (SRC);
  A_STORAGE := new STRING (1 .. LEN);
  C_START := SRC.all'ADDRESS;
  for INX in 0 .. LEN - 1 loop
    C_CUR := FETCH_CHAR (FROM => C_START + INTEGER (INX));
    A_STORAGE.all (INX + 1) := C_CUR;
  end loop;
  return A_STORAGE;
end CONVERT_TO_ADA;

begin -- Start of READ_C_STRING
  declare
    A_RESULT : A_STRING;
    C_RESULT : C_STRING;
  begin
    C_RESULT := SEND_C_STR;                -- Call the external C subprogram
    A_RESULT := CONVERT_TO_ADA( C_RESULT ); -- Convert to an access Ada STRING
    TEXT_IO.PUT_LINE( A_RESULT.all );      -- Print out the result.
  end;
end READ_C_STRING;
```

### F 11.3.4 Record Types and HP C Subprograms

See Section F 11.1.4 for details.

Ada records can be passed as parameters to external interfaced subprograms written in HP C if care is taken regarding the record layout and access to record discriminant values. See Section F 4.8, "Record Types," for information on record type layout.

## F 11.4 Calling HP FORTRAN 77 Language Subprograms

When calling interfaced HP FORTRAN 77 subprograms, the following form is used:

```
pragma INTERFACE(FORTRAN, Ada_subprogram_name)
```

This form is used to identify the need for HP FORTRAN 77 parameter passing conventions.

To call the HP FORTRAN 77 subroutine

```
Subroutine FSUB (Parm)
Integer*4 Parm
. . .
end
```

you need this interfaced subprogram declaration in Ada:

```
procedure FSUB (PARM : in out INTEGER);
pragma INTERFACE (FORTRAN, FSUB);
```

The external name specified in the Ada interface declaration can be any Ada identifier. If the Ada identifier differs from the FORTRAN 77 subprogram name, **pragma** INTERFACE\_NAME is required.

Note that the parameter in the example above is of mode **in out**. In HP FORTRAN 77, all user-declared parameters are always passed by reference; therefore, mode **in out** or mode **out** must be used for scalar type parameters. The HP FORTRAN 77 compiler might expect some implicit parameters that are passed by value and not by reference. See Section F 11.4.4, "String Types," for details.

Only scalar types (integer, floating point, and character types) are allowed for the result returned by an external interfaced function subprogram written in HP FORTRAN 77. Access type results are not supported.

For more information, see the following manuals:

- *HP FORTRAN 77 Reference Manual*
- *HP FORTRAN 77 Programmer's Guide*
- *HP FORTRAN 77 Quick Reference Guide*

For general information about passing types to interfaced subprograms, see Section F 11.1.

### F 11.4.1 Scalar Types and HP FORTRAN 77 Subprograms

FORTRAN expects all user-declared parameters to be passed by reference. Ada scalar type parameters will only be passed by reference if declared as `mode in out` or `out`; therefore, no scalar type parameters to a FORTRAN interface routine should be declared as `mode in`. No error will be reported by Ada, but you will most likely get unexpected results.

All Ada scalar type parameters are passed by reference to FORTRAN external subprograms. Scalar type parameters therefore must be declared as `mode in out` to be passed by reference.

#### F 11.4.1.1 Integer Types and HP FORTRAN 77 Subprograms

See Section F 11.1.1 for details.

Table F-13 summarizes the correspondence between integer types in Ada/800 and HP FORTRAN 77.

**Table F-13. Ada/800 versus HP FORTRAN 77 Integer Correspondence**

Ada	HP FORTRAN 77	Bit Length
SHORT_SHORT_INTEGER	BYTE	8
SHORT_INTEGER	INTEGER*2	16
INTEGER	INTEGER*4	32

The compatible types are the same for procedures and functions. Compatible Ada integer types are allowed for the result returned by an external interfaced function subprogram written in HP FORTRAN 77.

Ada semantics do not allow parameters of `mode in out` to be passed to function subprograms. Therefore, for Ada to call HP FORTRAN 77 external interfaced function subprograms, each scalar parameter's address must be passed. The use of the supplied package `SYSTEM` facilitates this passing of the object's address. The parameters in an HP FORTRAN 77 external function must be declared as in the example on the following page:



```

-- Ada declaration
with SYSTEM;
VAL1   : INTEGER; -- a scalar type
VAL2   : FLOAT ; -- a scalar type
RESULT : INTEGER;
function FTNFUNC ( PARM1, PARM2 : SYSTEM.ADDRESS) return INTEGER;

```

The external function must be called from within Ada as follows:

```

RESULT := FTNFUNC (VAL1'ADDRESS, VAL2'ADDRESS);

```

Because this has the effect of obscuring the types of the actual parameters, it is suggested that such declarations be encapsulated within an inlined Ada body so that the parameter types are made visible. An example follows:

```

-- specification of function to encapsulate
function FTNFUNC (PARM1 : INTEGER;
                  PARM2 : FLOAT   ) return INTEGER;
pragma INLINE (FTNFUNC);

with SYSTEM;
-- body of function to encapsulate
function FTNFUNC (PARM1 : INTEGER;
                  PARM2 : FLOAT   ) return INTEGER is
  function FORTFUNC (P1, P2 : SYSTEM.ADDRESS) return INTEGER;
  pragma INTERFACE (FORTRAN, FORTFUNC);

begin -- function FTNFUNC
  return FORTFUNC (PARM1'ADDRESS, PARM2'ADDRESS);
end FTNFUNC;

```

In the previous example, the name of the interfaced external function subprogram (written in HP FORTRAN 77) is FORTFUNC. This name is declared in the following way:

```

Integer*4 FUNCTION FORTFUNC (I, X)
Integer*4 I
Real*4 X
...
end

```

#### F 11.4.1.2 Enumeration Types and HP FORTRAN 77 Subprograms

The HP FORTRAN 77 language does not support enumeration types. However, objects that are elements of an Ada's enumeration type can be passed to an HP FORTRAN 77 integer type as the underlying representation of an enumeration type. The appropriate FORTRAN type (BYTE, INTEGER\*2, or INTEGER\*4) should be chosen to match the size of the Ada enumeration type. If a representation specification applies to the Ada enumeration type, the value specified by the representation clause (not the 'POS value) will be passed to the FORTRAN routine.

### F 11.4.1.3 Boolean Types and HP FORTRAN 77 Subprograms

See Section F 11.1.1.3 for details.

An Ada/800 Boolean that has the default 8-bit size is compatible with the default mode HP FORTRAN 77 type LOGICAL\*1 both as a parameter and as a function result.

An Ada/800 Boolean type with a representation specification for a larger size (16 or 32 bits) is *not* compatible with the larger sized HP FORTRAN 77 logical types (LOGICAL\*2 or LOGICAL\*4). Such Ada Booleans can be passed to the appropriately sized FORTRAN integer type (INTEGER\*2 or INTEGER\*4) and treated as integers that have the value of 'POS of the Ada Boolean value.

If the HP FORTRAN 77 routine is compiled with one of the HP FORTRAN 77 options that changes the size or representation of logical types to other than the default, you will have to determine what Ada types, if any, are compatible with the altered FORTRAN behavior by consulting the appropriate FORTRAN documentation.

### F 11.4.1.4 Character Types and HP FORTRAN 77 Subprograms

See Section F 11.1.1.4 for details.

There is no one-to-one mapping between an Ada character type and any HP FORTRAN 77 character type. An Ada character type can be passed to HP FORTRAN 77 or returned from HP FORTRAN 77 using one of several methods.

HP FORTRAN 77 considers all single character parameters to be single-element character arrays. The method that HP FORTRAN 77 uses to pass character arrays is described in Section F 11.4.4. The method requires that an implicit *value* parameter be passed to indicate the size of the character array. Because HP FORTRAN 77 uses this method for passing character types, it might be more convenient to convert Ada character types into Ada strings and follow the rules that govern passing Ada string types to HP FORTRAN 77.

An Ada/800 character that has the default 8-bit size can be passed to a default mode HP FORTRAN 77 parameter of type CHARACTER\*1. This can be done if the interface declaration specifies the additional size parameters that HP FORTRAN 77 implicitly expects and passes the constant value one (the size of the character) when the HP FORTRAN 77 subprogram is called. See Section F 11.4.4 for an example of implicit size parameters for strings; to pass an Ada character instead of a string, simply use the Ada character type in the Ada interface declaration in place of the Ada string type and CHARACTER\*1 in the HP FORTRAN 77 declaration in place of the CHARACTER \*(\*). Note that the size parameter(s) are not specified in the HP FORTRAN 77 subprogram declaration; they are implicit parameters that are expected by the HP FORTRAN 77 subprogram for each character array (or character) type parameter.

An Ada/800 character type that has the default size *cannot* be returned from an HP FORTRAN 77 function that has a result type of CHARACTER\*1 (it can be returned as a BYTE; see below for details).

An Ada/800 character type that has the default 8-bit size can also be passed to an HP FORTRAN 77 parameter of type BYTE without having to pass the additional length parameter. The BYTE will have the value of 'POS of the Ada character value.

An Ada/800 character type that has the default size can also be returned from an HP FORTRAN 77 function that has a return type of BYTE. The BYTE to be returned should be assigned the 'POS value of the desired Ada character.

An Ada/800 character type with a representation specification for a larger size (16 or 32 bits) is *not* compatible with any HP FORTRAN 77 character type. Such Ada characters can be passed to the appropriately sized FORTRAN integer type (INTEGER\*2 or INTEGER\*4) and treated as integers that have the value of 'POS of the Ada character value.

#### **F 11.4.1.5 Real Types and HP FORTRAN 77 Subprograms**

This section discusses passing fixed and floating point types to subprograms written in FORTRAN.

##### **Fixed Point Types**

Ada fixed point types are not supported as parameters or as results of external interfaced subprograms written in HP FORTRAN 77. Ada fixed point types *cannot* be returned as function results from external interfaced subprograms written in HP FORTRAN 77.

##### **Floating Point Types**

See Section F 11.1.1.5 for details.

The Ada type FLOAT corresponds to the REAL\*4 format in HP FORTRAN 77. The Ada type LONG\_FLOAT corresponds to the HP FORTRAN 77 type DOUBLE PRECISION (or REAL\*8).

There is no Ada type that corresponds to the HP FORTRAN 77 type REAL\*16.

#### **F 11.4.2 Access Types and HP FORTRAN 77 Subprograms**

Ada access types have no meaning in HP FORTRAN 77 subprograms because the types are address pointers to Ada objects. The implementation value of an Ada parameter of type ACCESS may be passed to an HP FORTRAN 77 procedure. The parameter in HP FORTRAN 77 is seen as INTEGER\*4. The object pointed to by the access parameter has no significance in HP FORTRAN 77; the access parameter value itself would be useful only for comparison operations to other access values.

HP FORTRAN 77 can return an INTEGER\*4 and the Ada program can declare an access type as the returned value type (it will be a matching size, because in Ada/800, an access type is a 32-bit quantity.) However, care should be taken that the returned value can actually be used by Ada in a meaningful manner.

### F 11.4.3 Array Types and HP FORTRAN 77 Subprograms

See Section F 11.1.3 for details.

Arrays whose components have an HP FORTRAN 77 representation can be passed as parameters between Ada and interfaced external HP FORTRAN 77 subprograms. For example, Ada arrays whose components are of types INTEGER, SHORT\_INTEGER, FLOAT, LONG\_FLOAT, or CHARACTER may be passed as parameters.

Array types *cannot* be returned as function results from external HP FORTRAN 77 subprograms. However, an access type to the array type can be returned as a function result.

#### CAUTION

Arrays with multiple dimensions are implemented differently in Ada and HP FORTRAN 77. To obtain the same layout of components in memory as a given HP FORTRAN 77 array, the Ada equivalent must be declared and used with the dimensions in reverse order.

Consider the components of a 2-row by 3-column matrix, declared in HP FORTRAN 77 as:

```
INTEGER*4 A(2,3) or INTEGER*4 A(1:2,1:3)
```

This array would be stored by HP FORTRAN 77 in the following order:

```
A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3)
```

This is referred to as storing in *column major order*; that is, the first subscript varies most rapidly, the second varies next most rapidly, and so forth, and the last varies least rapidly.

Consider the components of a 2-row by 3-column matrix, declared in Ada as:

```
A : array (1..2, 1..3) of INTEGER;
```

This array would be stored by Ada in the following order:

```
A(1,1), A(1,2), A(1,3), A(2,1), A(2,2), A(2,3)
```

This is referred to as storing in *row major order*; that is, the last subscript varies most rapidly, the next to last varies next most rapidly, and so forth, while the first varies least rapidly. Clearly the two declarations in the different languages are not equivalent. Now, consider the components of a 2-row by 3-column matrix, declared in Ada as:

```
A : array (1..3, 1..2) of INTEGER;
```

Note the reversed subscripts compared with the FORTRAN declaration. This array would be stored by Ada in the following order:

A(1,1), A(1,2), A(2,1), A(2,2), A(3,1), A(3,2)

If the subscripts are reversed, the layout would be

A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3)

which is identical to the HP FORTRAN 77 layout. Thus, either of the language declarations could declare its component indices in *reverse* order to be compatible.

To illustrate that equivalent multidimensional arrays require a reversed order of dimensions in the declarations in HP FORTRAN 77 and Ada, consider the following:

The Ada statement

FOO : array (1..10,1..5,1..3) of FLOAT;

is equivalent to the HP FORTRAN 77 declaration:

REAL\*4 FOO(3,5,10)

or

REAL\*4 FOO(1:3,1:5,1:10)

Both Ada and HP FORTRAN 77 store a one-dimensional array as a linear list.

#### F 11.4.4 String Types and HP FORTRAN 77 Subprograms

When a string item is passed as an argument to an HP FORTRAN 77 subroutine from within HP FORTRAN 77, extra information is transmitted in hidden (implicit) parameters. The calling sequence includes a hidden parameter (for each string) that is the actual length of the ASCII character sequence. This implicit parameter is passed in addition to the address of the ASCII character string. The hidden parameter is passed by value, not by reference.

These conventions are different from those of Ada. For an Ada program to call an external interfaced subprogram written in HP FORTRAN 77 with a string type parameter, you must explicitly pass the length of the string object. The length must be declared as an Ada 32-bit integer parameter of mode in.

## Implementation-Dependent Characteristics

The following example illustrates the declarations needed to call an external subroutine having a parameter profile of two strings and one floating point variable.

```
procedure FTNSTR is
  SA: STRING(1..6):= "ABCDEF";
  SB: STRING(1..2):= "GH";
  FLOAT_VAL: FLOAT:= 1.5;
  LENGTH_SA, LENGTH_SB : INTEGER;

  procedure FEXSTR ( S1 : STRING;           -- passed by reference
                    LS1 : in INTEGER ;      -- len of string S1,
                                           -- must be IN
                    F   : in out FLOAT;    -- must be IN OUT
                    S2 : STRING;           -- passed by reference
                    LS2 : in INTEGER);      -- len of string S2,
                                           -- must be IN

  pragma INTERFACE (FORTRAN, FEXSTR);

begin -- procedure FTNSTR
  LENGTH_SA := SA'LENGTH;
  LENGTH_SB := SB'LENGTH;
  FEXSTR (SA, LENGTH_SA, FLOAT_VAL, SB, LENGTH_SB);
end FTNSTR;
```

### NOTE

Note that the string lengths immediately follow the corresponding string parameter. The string lengths must be passed by value, *not* by reference.

The HP FORTRAN 77 external subprogram is the following:

```
SUBROUTINE FEXTR (S1, r, S2)
CHARACTER *(*) S1, S2
REAL*4 r
...
END
```

### NOTE

Ada/800 does *not* allow a string type (constrained or *not*) to be returned from a function interfaced with HP FORTRAN 77. Thus, it is *not* possible to declare an Ada external interfaced function that returns a result of type STRING (type STRING is an object of type CHARACTER \*(\*) in HP FORTRAN 77).

### F 11.4.5 Record Types and HP FORTRAN 77 Subprograms

See Section F 11.1.4 for details.

Ada records may be passed as parameters to external interfaced subprograms written in HP FORTRAN if care is taken regarding the record layout and access to record discriminant values. See Section F 4.8, "Record Types," for information on record type layout.

Record types are *not* allowed as function results in HP FORTRAN functions.

### F 11.4.6 Other FORTRAN Types

The HP FORTRAN 77 types COMPLEX, COMPLEX\*8, DOUBLE COMPLEX, and COMPLEX\*16 have no direct counterparts in Ada. However, it is possible to declare equivalent types using either an Ada array or an Ada record type. For example, with type COMPLEX in HP FORTRAN 77, a simple Ada equivalent is a user-defined record:

```
type COMPLEX is
  record
    Real : FLOAT;
    Imag : FLOAT;
  end record;
```

Similarly, an HP FORTRAN 77 double complex number could be represented with the two record components declared as Ada type LONG\_FLOAT.

While it is *not* possible to declare an Ada external function that returns the above record type, an Ada procedure *can* be declared with an out parameter of type COMPLEX. The Ada procedure would then need to interface with an HP FORTRAN 77 subroutine, which would pass the result back using an in out or out parameter.

## F 11.5 Calling HP Pascal Language Subprograms

When calling interfaced HP Pascal subprograms, the form

```
pragma INTERFACE (Pascal, Ada_subprogram_name)
```

is used to identify the need to use the HP Pascal parameter passing conventions.

To call the following HP Pascal subroutine

```
module modp;
  export
    procedure p_subr ( val_parm : integer;
                      var ref_parm : integer );

  implement
    procedure p_subr ( val_parm : integer;
                      var ref_parm : integer );
  begin
    . . .
  end;
end.
```

Ada would use the interfaced subprogram declaration:

```
procedure P_SUB (VAL_PARAM : in INTEGER;
                 REF_PARAM : in out INTEGER);
pragma INTERFACE (Pascal, P_SUB);
```

In the above example we provided the Ada subprogram identifier P\_SUB to the pragma INTERFACE.

Note that the parameter in the example, VAL\_PARAM, must be of mode *in*, to match the parameter definition for val\_parm found in the HP Pascal subroutine. Likewise, REF\_PARAM, must be of mode *in out* to correctly match the HP Pascal definition of var ref\_parm. Also, note that the names for parameters *do not* need to match exactly. However, the mode of access and the data type *must* be correctly matched, but there is no compile-time or run-time check that can ensure that they match. It is your responsibility to ensure their correctness.

When Ada interfaces to HP Pascal, it refers to the HP Pascal procedure or function by the procedure or function name. In the above example, the pragma INTERFACE was sufficient to specify that name, although a pragma INTERFACE\_NAME could also have been used (and would be necessary if the name given to the Ada routine did not map correctly to the desired HP Pascal name). Because Ada uses only the HP Pascal procedure or function name, there is a difficulty if that name is not unique.

The names of the procedures and functions declared within an HP Pascal module must be unique within a single module. A given module can only contain one procedure or function name (for example, FOO), but another module could also contain a procedure or function named FOO. If a single program uses both modules, it is necessary to properly resolve references to FOO. To properly resolve such references, procedure and function names in modules are qualified with the name of the module that contains them. This qualification is internal to the object file and is not accessible to user code. The linker (*ld(1)*) uses the qualification information to resolve references by HP Pascal code to identically named procedures or functions.



This qualification mechanism poses a difficulty when attempting to interface Ada to HP Pascal because Ada can only specify the unqualified HP Pascal procedure or function name. There will be no difficulty if the HP Pascal procedure or function being called has a name that is unique within all the HP Pascal modules used in the Ada program (if the qualification mechanism is not needed for the name). If the procedure or function name is not unique, the linker (*ld(1)*) will, without producing an error or warning, select one (usually the first one) of the multiple HP Pascal procedures or functions that it encounters during the link that has the name specified by Ada. As this unpredictable selection is likely to lead to an incorrect program, interfacing to HP Pascal procedures or functions that are not uniquely named is not recommended.

For more information on Pascal interfacing, see the *HP Pascal Language Reference Manual*. Additional information is available in the *HP-UX Portability Guide*.

For Pascal, scalar and access parameters of mode *in* are passed by value; the value of the parameter object is copied and passed. All other types of *in* parameters (arrays and records) and parameters of mode *out* and *in out* are passed by reference; the address of the object is passed. This means that, in general, Ada *in* parameters correspond to Pascal value parameters, while Pascal *var* parameters correspond to the Ada parameters of either mode *in out* or mode *out*.

Only scalar types (integer, floating point, character, Boolean, and enumeration types) and access types are allowed for the result returned by an external interfaced Pascal function subprograms.

For general information about passing parameters to interfaced subprograms, see Section F 11.1.

### F 11.5.1 Scalar Types and HP Pascal Subprograms

See Section F 11.1.1 for details.

#### F 11.5.1.1 Integer Types and HP Pascal Subprograms

Integer types are compatible between Ada and HP Pascal provided their ranges of values are identical. Table F-14 shows corresponding integer types in Ada and HP Pascal.

Table F-14. Ada/800 versus HP Pascal Integer Correspondence

Ada	HP Pascal	Bit Length
predefined type INTEGER	predefined type integer	32
predefined type SHORT_INTEGER	predefined type shortint or user type I16 = 0..65535;	16
predefined type SHORT_SHORT_INTEGER	user-defined type type I8 = 0..255;	8

**NOTE**

In HP Pascal, any integer subrange that has a negative lower bound is always implemented in 32 bits. Integer subranges with a non-negative lower bound are implemented in 8-bits if the upper bound is 255 or less, in 16-bits if the upper bound is 65535 or less, and in 32-bits if the upper bound is greater than 65535. Therefore, in Table F-14 above, the user-defined subrange 0..255 is shown as the HP Pascal equivalent to the Ada type `SHORT_SHORT_INTEGER`; however, the Ada type is a signed type with the range  $-128..127$ . To convert the unsigned value back to a signed value in HP Pascal, if the unsigned value is greater than 127, you will need to subtract 256 to obtain the actual negative value.

Whether passed from Ada to HP Pascal by value or by reference, the appropriate HP Pascal type, from Table F-14 above, must be used to properly access the Ada integer value from HP Pascal.

All Ada integer types are allowed for the result returned by an external interfaced subprogram written in HP Pascal if care is taken with respect to ranges defined for integer quantities.

### F 11.5.1.2 Enumeration Types and HP Pascal Subprograms

See Section F 11.1.1.2 for details.

Ada and HP Pascal have similar implementations of enumeration types. In Ada and HP Pascal, enumeration types can have a size of 8, 16, or 32 bits. However, Ada normally considers enumeration types to be signed quantities and HP Pascal considers them to be unsigned. Table F-15 shows corresponding enumeration types in Ada and HP Pascal.

**Table F-15 Ada/800 versus HP Pascal Enumeration Correspondence**

Ada	HP Pascal	Bit Length
$\leq 128$ elements	$\leq 256$ elements	8
$\leq 32768$ elements	$\leq 65536$ elements	16
$> 32768$ elements	$> 65536$ elements	32

If the Ada enumeration type has 129 through 256 elements or 32769 through 65536 elements, there are additional requirements to passing or returning values of such an Ada type. A size specification on a representation clause for the Ada enumeration type should be used to specify the minimum size for the enumeration type (see Section F 4.1 for details.) When such a size is used and none of the internal codes are negative integers, the internal representation of the Ada type will be unsigned and will conform with the HP Pascal representation.

If such a size specification representation clause is not used, it is still possible to pass a simple variable or expression of such a type to HP Pascal, by value or reference, or to return one from HP Pascal. Although the Ada enumeration object is stored in a larger container than HP Pascal expects, the valid values are actually all stored within the part of the container that HP Pascal will access.

However, unless a size specification representation clause is used, there will be difficulty passing arrays of Ada enumeration values of such types or passing records containing fields of such types. HP Pascal will not properly access the correct elements of such arrays or fields of such records because it will assume the enumeration values to be smaller than they actually are and therefore will compute their location incorrectly.

If a representation specification is applied to the Ada enumeration type to alter the internal value of any enumeration elements, care must be taken that the values are within the HP Pascal enumeration type to which the Ada enumeration value is being passed.

Ada supports the return of a function result that is an enumeration type from an external interfaced function subprogram written in HP Pascal.

### **F 11.5.1.3 Boolean Types and HP Pascal Subprograms**

See Section F 11.1.1.3 for details.

### **F 11.5.1.4 Character Types and HP Pascal Subprograms**

See Section F 11.1.1.4 for details.

Values of the Ada predefined character type might be treated as the type CHAR in HP Pascal external interfaced subprograms.

### **F 11.5.1.5 Real Types and HP Pascal Subprograms**

The following subsections discuss passing Ada real types to interfaced HP Pascal subprograms.

#### **Fixed Point Types**

Ada fixed point types are not supported as parameters or as results of external subprograms. Ada fixed point types *cannot* be returned as function results from interfaced subprograms written in HP Pascal.

## Implementation-Dependent Characteristics

### Floating Point Types

See Section F 11.1.1.5 for details.

Ada FLOAT values correspond to HP Pascal real values. Ada LONG\_FLOAT values correspond to HP Pascal longreal values.

### F 11.5.2 Access Types and HP Pascal Subprograms

See Section F 11.1.2 for details.

Ada access values can be treated as pointer values in HP Pascal. The Ada heap allocation and the HP Pascal heap allocation are completely separate. There must be no explicit deallocation of an access/pointer object in one language of an object allocated in the other language.

### F 11.5.3 Array Types and HP Pascal Subprograms

See Section F 11.1.3 for details.

Arrays with components with the same representation have the same representation in Ada and HP Pascal.

Arrays *cannot* be passed by value from Ada to HP Pascal. An Ada array can only be passed to a VAR parameter in an HP Pascal subprogram.

Array types *cannot* be returned as function results from external interfaced subprograms written in HP Pascal.

### F 11.5.4 String Types and HP Pascal Subprograms

See Section F 11.1.3 for details.

Passing variable length strings between Ada and HP Pascal is supported with some restrictions. Strings *cannot* be passed by value from Ada to HP Pascal. An Ada string can only be passed to a VAR parameter in an HP Pascal subprogram.

String types *cannot* be returned as function results from external HP Pascal subprograms.

Although there is a difference in the implementation of the type STRING in the two languages, with suitable declarations you can create compatible types to allow the passing of both Ada strings and HP Pascal strings. An Ada string corresponds essentially to a packed array of characters in Pascal. However, the Ada string type must be one character longer than the corresponding string type in the HP Pascal procedure or function. HP Pascal adds such an implicit extra byte to its own packed arrays of characters and expects to be able to utilize this extra byte during some string operations. The following example illustrates the declaration of compatible types for passing an Ada string between an Ada program and an HP Pascal subprogram.

HP Pascal subprogram:

```
(* passing an Ada STRING type to an HP Pascal routine *)
module p;
export
  type string80 = packed array [1..80] of char;
  procedure ex1 ( var s : string80; len : integer );
implement
  procedure ex1;
  begin
    ... (* update/use the Ada string as a PAC *)
  end;
end.
```

## Implementation-Dependent Characteristics

### Ada program:

```
-- Ada calling HP Pascal procedure with Ada STRING
procedure AP_1 is

    -- Define Ada string corresponding to HP Pascal packed array of char
    subtype STRING80 is STRING ( 1..81 ); -- 80+1 for HP Pascal

    -- Ada definition of HP Pascal procedure to be called, with an
    -- Ada STRING parameter, passed by reference.
    procedure EX1 (S      : in out STRING80;
                  LEN    :      INTEGER );
    pragma INTERFACE (PASCAL, EX1);
    pragma INTERFACE_NAME (EX1, "ex1");

    S      : STRING80;

begin -- AP_1
    S(1..26) := "Ada to HP Pascal Interface";
    EX1 (S, 26);      -- Call the HP Pascal subprogram
end AP_1;
```

An HP Pascal STRING type corresponds to a record in Ada that contains two fields: a 32-bit integer field containing the string length and an Ada STRING field containing the string value. The following example illustrates the declaration of compatible types for passing an HP Pascal string between an Ada program and a Pascal subprogram.

**Pascal subprogram:**

```
(* passing an HP Pascal STRING type from Ada to an HP Pascal routine *)
module p;
export
  type string80 = string[80];
  procedure ex2 ( var s : string80 );
implement

  procedure ex2;
  var
    str : string80 ;
  begin
    ... --update/use the HP Pascal string
  end;
end.
```

**Ada program:**

```
-- Ada calling HP Pascal procedure using a HP Pascal string[80]
procedure AP_2 is

  -- Define an Ada record that will correspond exactly
  -- with the HP Pascal type: string[80]

  type PASCAL_STRING80 is
    record
      LEN : INTEGER;
      S   : STRING ( 1..81 ); -- 80+1 for HP Pascal
    end record;

  -- Here we use a record representation clause to
  -- force the compiler to layout the record in
  -- the correct manner for HP Pascal

  for PASCAL_STRING80 use
    record
      LEN at 0 range 0 .. 31;
      S   at 1 range 0 .. 81*8; -- 80+1 for HP Pascal
    end record;
```

## Implementation-Dependent Characteristics

```
-- The Ada definition of the HP Pascal procedure to be
-- called, with an HP Pascal STRING parameter, passed
-- by reference.

procedure EX2 (S : in out PASCAL_STRING80);
pragma INTERFACE (PASCAL, EX2);
pragma INTERFACE_NAME (EX2, "ex2");

PS      : PASCAL_STRING80;

begin -- AP_2

    PS.S(1..26) := "Ada to HP Pascal Interface"; -- assign value field
    PS.LEN := 26; -- set string length field
    EX2 ( PS ); -- call the HP Pascal subprogram

end AP_2;
```

### F 11.5.5 Record Types and HP Pascal Subprograms

See Section F 11.1.4 for details.

Records *cannot* be passed by value from Ada to HP Pascal. An Ada record can only be passed to a VAR parameter in an HP Pascal subprogram.

Record types *cannot* be returned as function results from external HP Pascal subprograms.



**F 11.6 Summary**

Table F-16 shows how various Ada types are passed to subprograms.

**Table F-16. Modes for Passing Parameters to Interfaced Subprograms**

Ada Type	Mode	Passed By
ACCESS, SCALAR -INTEGER -ENUMERATION -BOOLEAN -CHARACTER -REAL	in	value
ARRAY, RECORD	in	reference
all types except TASK and FIXED POINT	in out	reference
all types except TASK and FIXED POINT	out	reference
TASK FIXED POINT	N/A	not passed

## Implementation-Dependent Characteristics

Table F-17 summarizes general information presented in Section F 11.1.

**Table F-17. Types Returned as External Function Subprogram Results**

Ada Type	HP-PA Assembler	HP C	HP FORTRAN	HP Pascal
INTEGER	allowed	allowed	allowed	allowed
ENUMERATION	allowed	allowed	not allowed <sup>1</sup>	allowed
CHARACTER	allowed	allowed	not allowed	allowed
BOOLEAN	allowed	allowed	allowed	allowed
FLOAT	allowed	allowed <sup>2</sup>	allowed	allowed
FIXED POINT	not allowed	not allowed	not allowed	not allowed
ACCESS	allowed	allowed	not allowed <sup>1</sup>	allowed
ARRAY	not allowed	not allowed	not allowed	not allowed
STRING	not allowed	not allowed	not allowed	not allowed
RECORD	not allowed	not allowed	not allowed	not allowed
TASK	not allowed	not allowed	not allowed	not allowed

**Notes for Table F-19:**

<sup>1</sup> Pass as an integer equivalent.

<sup>2</sup> Some restrictions apply to Ada FLOAT types (in passing to HP C subprograms).

Table F-18 summarizes information presented in Sections F 11.2 through F 11.5.

**Table F-18. Parameter Passing in the Series 800 Implementation**

Ada Type	HP-PA Assembler	HP C	HP FORTRAN	HP Pascal
INTEGER	allowed	allowed	allowed	allowed
ENUMERATION	allowed	allowed	not allowed <sup>1</sup>	allowed
CHARACTER	allowed	allowed	not allowed <sup>2</sup>	allowed
BOOLEAN	allowed	allowed	not allowed <sup>1</sup>	allowed
FLOAT	allowed	allowed	allowed	allowed
FIXED POINT	not allowed	not allowed	not allowed	not allowed
ACCESS	allowed	allowed	not allowed	allowed
ARRAY <sup>3</sup>	allowed	allowed	allowed <sup>4</sup>	allowed
STRING	allowed	allowed <sup>5</sup>	allowed <sup>6</sup>	not allowed <sup>7</sup>
RECORD	allowed	allowed	allowed	allowed
TASK	not allowed	not allowed	not allowed	not allowed

**Notes for Table F-20:**

<sup>1</sup> Can be passed as an equivalent integer value

<sup>2</sup> Must be passed as a STRING.

<sup>3</sup> Using only arrays of compatible component types.

<sup>4</sup> See warning on layout of elements.

<sup>5</sup> Special handling of null terminator character is required.

<sup>6</sup> Requires that the length also be passed.

<sup>7</sup> Ada strings can be passed to a Pascal PAC (Packed Array of Characters)

## F 11.7 Potential Problems Using Interfaced Subprograms

The Ada runtime for the HP 9000 Series 800 computer uses signals in a manner that generally does not interfere with interfaced subprograms. However, some HP-UX routines are interruptible by signals. These routines, if called from within interfaced external subprograms, may create problems. You need to be aware of these potential problems when writing external interfaced subprograms in other languages that will be called from within an Ada main subprogram. See *sigvector(2)* in the *HP-UX Reference* for a complete explanation of interruptibility of operating system routines.

The following should be taken into consideration:

- SIGALRM is sent when a **delay** statement is being timed in a tasking program.
- SIGVTALRM is sent when round-robin scheduling is used in a tasking program.
- Interruptible HP-UX routines (see *sigvector(2)*) may need to be protected from interruption by the signals used by the Ada runtime system. The `SYSTEM_ENVIRONMENT` routines `SUSPEND_ADA_TASKING` and `RESUME_ADA_TASKING` can be used to implement this protection. As an alternative, the knowledgeable user can use the *sigsetmask(2)* or *sigblock(2)* mechanism to implement the same protection.
- If a signal is received while it is blocked, one instance of the signal is guaranteed to remain pending and will be honored when the signal is unblocked. Any additional instances of the signal will be lost.
- Any signals blocked in interfaced code should be unblocked before leaving the interfaced code.

The SIGALRM and SIGVTALRM signals (noted above) are the most likely signals to cause problems with interfaced subprograms. They are asynchronous signals; that is, they can occur at any time and they are not caused by the code that is executing at the time they occur. In addition, SIGALRM might unexpectedly interrupt HP-UX (or other) routines that are sensitive to being interrupted by signals.

Problems can arise if an interfaced subprogram initiates a "slow" operating system function that can be interrupted by a signal (for example, a *read(2)* call on a terminal device or a *wait(2)* call that waits for a child process to complete). Problems can also arise if an interfaced subprogram can be called by more than one task and is not reentrant. If an Ada reserved signal occurs during such an operation or non-reentrant region, the program may function erroneously.

For example, an Ada program that uses **delay** statements and tasking constructs causes the generation of SIGALRM and SIGVTALRM. If an interfaced subprogram needs to perform a potentially interruptible operating system call, or if it might be called from more than one task and is not reentrant, it can be protected by blocking SIGALRM and SIGVTALRM around the operating system call or non-reentrant region. If a SIGALRM or SIGVTALRM signal signifying either the end of a delay period or the need to reschedule a task is received while it is blocked, the signal is not lost, but rather deferred until it is later unblocked. The consequence of this signal blocking is that Ada task scheduling or **delay** statement execution will be affected for the duration of the signal block.

Here is an example of a protected *read(2)* in an interfaced subprogram written in the C language.

```
#include <signal.h>
void interface_rout()
{
    long mask;

    ...

    /* Add SIGALRM and SIGVTALRM to list of currently
       blocked signals. (see sigblock(2)). */

    mask = sigblock (( 1L << (SIGALRM-1)) | (1L << (SIGVTALRM-1)));

    ...    read (...) ;    /* or non reentrant region */

    sigsetmask (mask) ;    /* return to previous mask */
}
```

If any Ada reserved signal other than SIGALRM or SIGVTALRM is to be similarly blocked, SIGALRM and SIGVTALRM must be either already blocked or blocked at the same time. When any Ada reserved signal other than SIGALRM or SIGVTALRM is unblocked, SIGALRM and SIGVTALRM must be unblocked at the same time, or as soon as possible thereafter.

Any Ada reserved signal blocked in interfaced code should be unblocked before leaving that code, or as soon as possible thereafter, to avoid unnecessarily stalling the Ada runtime executive. Failure to follow these guidelines will cause improper **delay** or tasking operation.

An alternative method of protecting interfaced code from signals is described in the *Ada User's Guide* in the section on "Execution-Time Topics." The two procedures `SUSPEND_ADA_TASKING` and `RESUME_ADA_TASKING` from the package `SYSTEM_ENVIRONMENT` supplied by Hewlett-Packard can be used within an Ada program to surround a critical section of Ada code or a call to external interfaced subprogram code with a critical section.

## F 11.8 Input-Output From Interfaced Subprograms

Using I/O from interfaced subprograms written in other languages requires caution. Some areas in which problems can arise are discussed in this section.

### F 11.8.1 Files Opened by Ada and Interfaced Subprograms

An interfaced subprogram should *not* attempt to perform I/O operations on files opened by Ada. Your program should not use HP-UX I/O utilities intermixed with Ada I/O routines on the same file. If it is necessary to perform I/O operations in interfaced subprograms using the HP-UX utilities, open and close those files with HP-UX utilities.

### **F 11.8.2 Preconnected I/O and Interfaced Subprograms**

The standard HP-UX files `stdin` and `stdout` are preconnected by Ada I/O. If non-blocking interactive I/O is used, additional file descriptors will be used for interactive devices connected to `stdin` or `stdout`. Ada does not preconnect `stderr`, which is used for run-time error messages. An Ada subprogram called `PUT_TO_STANDARD_ERROR` is provided in the package `SYSTEM_ENVIRONMENT` which allows your program to output a line to the HP-UX stream `stderr`. For more details on Ada I/O, see the *Ada RM*, Section 14 and the section on using the Ada Development System in the *Ada User's Manual*.

### **F 11.8.3 Interactive I/O and Interfaced Subprograms**

The default I/O system behavior is `NON-BLOCKING` for Ada programs with tasking and `BLOCKING` for sequential (non-tasking) Ada programs. HP's implementation of Ada/800 will set non-blocking I/O by default for interactive files and pipes if the program contains tasks. If the Ada program contains no task structures (that is, it is a sequential program), blocking I/O is set for interactive files and pipes. You can override the defaults with binder options.

The binder option `-W b, -b` sets up blocking I/O and the binder option `-W b, -B` sets up non-blocking I/O. In non-blocking I/O, a task (or Ada main program) will not block when attempting interactive input if data is not available. If the I/O request cannot be immediately satisfied, the Ada runtime will place the task that requested I/O on a suspend queue and will awaken the task when the I/O operation is complete. This arrangement allows other tasks to continue execution; the task requesting I/O will be suspended until the I/O operation is completed by the Ada runtime.

# APPENDIX C TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
\$ACC SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
\$BIG ID1 An identifier the size of the maximum input line length which is identical to \$BIG_ID2 except for the last character.	(1..254 => 'A', 255 => '1')
\$BIG ID2 An identifier the size of the maximum input line length which is identical to \$BIG_ID1 except for the last character.	(1..254 => 'A', 255 => '2')
\$BIG ID3 An identifier the size of the maximum input line length which is identical to \$BIG_ID4 except for a character near the middle.	(1..127 => 'A', 128 => '3', 129..255 => 'A')

# TEST PARAMETERS

Name and Meaning	Value
<b>\$BIG_ID4</b> An identifier the size of the maximum input line length which is identical to \$BIG_ID3 except for a character near the middle.	(1..127 => 'A', 128 => '4', 129..255 => 'A')
<b>\$BIG_INT_LIT</b> An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..252 => '0', 253..255 => "298")
<b>\$BIG_REAL_LIT</b> A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..249 => '0', 250..255 => "69.0E1")
<b>\$BIG_STRING1</b> A string literal which when catenated with \$BIG_STRING2 yields the image of \$BIG_ID1.	(1 => '"', 2..128 => 'A', 129 => '"')
<b>\$BIG_STRING2</b> A string literal which when catenated to the end of \$BIG_STRING1 yields the image of \$BIG_ID1.	(1 => '"', 2..128 => 'A', 129 => '1', 130 => '"')
<b>\$BLANKS</b> A sequence of blanks twenty characters less than the size of the maximum line length.	(1..235 => ' ')
<b>\$COUNT_LAST</b> A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
<b>\$DEFAULT_MEM_SIZE</b> An integer literal whose value is SYSTEM.MEMORY_SIZE.	2147483647
<b>\$DEFAULT_STOR_UNIT</b> An integer literal whose value is SYSTEM.STORAGE_UNIT.	8



# TEST PARAMETERS

Name and Meaning	Value
\$DEFAULT_SYS NAME The value of the constant SYSTEM.SYSTEM_NAME.	HP9000_800
\$DELTA DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
\$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
\$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	SHORT_SHORT_FLOAT
\$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100000.0
\$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	100000000.0
\$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	127
\$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	not_there//not_there/*
\$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	not_there/not_there/not_there/ ../not_there777
\$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648

# TEST PARAMETERS

Name and Meaning	Value
\$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
\$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
\$LESS THAN DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100000.0
\$LESS THAN DURATION BASE FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-100000000.0
\$LOW PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	1
\$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
\$MAX DIGITS Maximum digits supported for floating-point types.	15
\$MAX IN LEN Maximum input line length permitted by the implementation.	255
\$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
\$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
\$MAX LEN INT BASED LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be \$MAX_IN_LEN long.	(1..2 => "2:", 3..252 => '0', 253..255 => "11:")

# TEST PARAMETERS

Name and Meaning	Value
<p><b>\$MAX_LEN_REAL_BASED_LITERAL</b>  A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be \$MAX_IN_LEN long.</p>	(1..3 => "16:", 4..251 => '0', 252..255 => "F.E:")
<p><b>\$MAX_STRING_LITERAL</b>  A string literal of size \$MAX_IN_LEN, including the quote characters.</p>	(1 => '"', 2..254 => 'A', 255 => '"')
<p><b>\$MIN_INT</b>  A universal integer literal whose value is SYSTEM.MIN_INT.</p>	-2147483648
<p><b>\$MIN_TASK_SIZE</b>  An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.</p>	32
<p><b>\$NAME</b>  A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.</p>	SHORT_SHORT_INTEGER
<p><b>\$NAME_LIST</b>  A list of enumeration literals in the type SYSTEM.NAME, separated by commas.</p>	HP9000_800
<p><b>\$NEG_BASED_INT</b>  A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.</p>	16#FFFFFFFD#
<p><b>\$NEW_MEM_SIZE</b>  An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than \$DEFAULT_MEM_SIZE. If there is no other value, then use \$DEFAULT_MEM_SIZE.</p>	1048576

# TEST PARAMETERS

Name and Meaning	Value
<p><b>\$NEW STOR UNIT</b>            An integer literal whose value is a permitted argument for pragma STORAGE UNIT. other than \$DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.</p>	8
<p><b>\$NEW SYS NAME</b>            A value of the type SYSTEM.NAME, other than \$DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.</p>	HP9000_800
<p><b>\$TASK SIZE</b>            An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.</p>	32
<p><b>\$TICK</b>            A real literal whose value is SYSTEM.TICK.</p>	0.01

APPENDIX D  
WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

- a. E28005C: This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this text that must appear at the top of the page.
- b. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).
- c. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).
- d. C97116A: This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING OF THE GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.
- e. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).
- f. CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

## WITHDRAWN TESTS

- g. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.
- h. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).
- i. CD2B15C and CD7205C: These tests expect that a 'STORAGE SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.
- j. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.
- k. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).
- l. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.
- m. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).
- n. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.
- o. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

## WITHDRAWN TESTS

- p. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA\_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).
- q. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.
- r. CE3301A: This test contains several calls to END\_OF\_LINE and END\_OF\_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD\_INPUT (lines 103, 107, 118, 132, and 136).
- s. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT\_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

## APPENDIX E

### COMPILER OPTIONS AS SUPPLIED BY THE HEWLETT PACKARD COMPANY

Compiler: HP 9000 Series 800 Ada Compiler, Version 4.35

ACVC Version: 1.10



## NAME

*ada* - Ada compiler

## SYNOPSIS

*ada* [ *options* ] [ *files* ] *libraryname*

## Remarks:

This command requires installation of optional Ada software (not included with the standard HP-UX operating system) before it can be used.

## DESCRIPTION

*Ada* is the HP-UX Ada compiler. It accepts several types of file arguments:

- (1) Arguments whose names end with *.ad?*, where *?* is any single alphanumeric character, are taken to be Ada source files.
- (2) The *libraryname* argument names an Ada library that must have been previously created using the *ada.mklib(1)* command. An Ada library is an HP-UX directory containing files that are used by the various components of the Ada compilation system. There is no required or standard suffix on the name of an Ada library.

The named source files are compiled, and each successfully compiled unit is placed in the specified Ada library by the compiler. When binding, or binding and linking, information is extracted from the Ada library to perform the bind and/or link operation. The Ada library must always be specified.

To ensure the integrity of the internal data structures of Ada libraries, libraries are locked for the duration of any operations which are performed on them. During compilation, *libraryname* is normally locked for updating and other Ada libraries can be locked for reading. During binding/linking, Ada libraries are only locked for reading.

If *ada* cannot obtain a lock after a suitable number of retries, it displays an informational message and terminates.

Users are strongly discouraged from placing any additional files in Ada library directories. User files in Ada libraries are subject to damage by or might interfere with proper operation of *ada* and related tools.

- (3) All other file arguments, including those whose names end with *.o* or *.a* are passed on to the linker *ld(1)* to be linked into the final program. It is not possible to link-only with the *ada(1)* command.
- (4) Although shown in the preferred order: above, *options*, *files*, and *libraryname* arguments can appear in any order.

## Environment Variables

The environment variable *ADA\_PATH* is associated with all components of the Ada compilation system. It must be set properly and exported before any component of the Ada compilation system (including *ada*) can be used (see *ada(1)*, Environment Variables).

Normally this variable is defined and set in the systemwide shell startup files */etc/profile* (for *sh(1)* and *ksh(1)*) and */etc/csh.login* (for *csh(1)*). However, it can be set by a user either interactively or in a personal shell startup file, *.profile* (for *sh(1)* and *ksh(1)*) or *.cshrc* (for *csh(1)*).

*ADA\_PATH* must contain the path name of the directory in which the Ada compiler components have been installed.

The value of this variable must be a rooted directory (i.e. it must begin with a */*) and the directory specification must not end with a */*.

**ADAOPTS**

The environment variable **ADAOPTS** can be used to supply commonly used (or default) arguments to *ada*. **ADAOPTS** is associated directly with *ada* (it is not used by any other component of the Ada compilation system).

Arguments can be passed to *ada* through the **ADAOPTS** environment variable, as well as on the command line. *Adu*(1) picks up the value of **ADAOPTS** and places its contents before any arguments on the command line. For example (in *sh*(1) notation),

```
$ ADAOPTS="-v -e 10"
$ export ADAOPTS
$ ada -L source.ada test.lib
```

is equivalent to

```
$ ada -v -e 10 -L source.ada test.lib
```

**Compiler Options**

The following options are recognized:

- a Store the supplied annotation string in the library with the compilation unit. This string can later be displayed by the unit manager. The maximum length of this string is 80 characters. The default is no string.
- b Display abbreviated compiler error messages (default is to display the long forms).
- c Suppress link phase and, if binding occurred, preserve the object file produced by the binder. This option only takes effect if linking would normally occur. Linking normally occurs when binding has been requested.

Use of this option causes an informational message to be displayed on standard error indicating the format of the *ld*(1) command that should be used to link the program. It is recommended that the user supply additional object (.o) and archive (.a) files and additional library search paths (-L) only in the places specified by the informational message.

If the -c option is given along with the -d or -D option, the binder must assume the name for the executable file, in order to determine what to name the debug information file (see -d and -D). If the -o option is not given, the debug information file will be named *a.out.cui*, and the user must ensure that the executable is named *a.out*. If a -o *outfile* option is given, the debug information file will be named *outfile.cui*, and the user must ensure that the executable is named *outfile*. If the executable is not named as expected, *ada.probe*(1) will not work properly.

When *ld* is later used to actually link the program, the following conditions must be met:

1. The .o file generated by the binder must be specified before any HP-UX archive is specified (either explicitly or with -l).
2. If -lc is specified when linking, any .o file containing code that uses *stdio*(3S) routines must be specified before -lc is specified.

- d Cause the compiler to store additional information in the Ada library for the units being compiled for use by the Ada debugger (see *ada.probe*(1)). Only information required for debugging is saved; the source is not saved. (see -D.) By default, debug information is not stored.

Cause the binder to produce a debug information file for the program being bound so that the resulting program can be manipulated by the Ada debugger. The debug

information file name will be the executable program file name with `.cul` appended. If the debug information file name would be truncated by the file system on which it would be created, an error will be reported.

Only sources compiled with the `-d` or `-D` option contribute information to the debug information file produced by the binder.

- `-e <nnn>` Stop compilation after `<nnn>` errors (legal range 0..32767, default 50).
- `-l` Cause any pending or existing instantiations of generic bodies in this Ada library, whose actual generic bodies have been compiled or recompiled in another Ada library, to be compiled (or recompiled) in this Ada library.  
  
This option is treated as a special "source" file and the compilation is performed when the option is encountered among the names of any actual source files.  
  
Any pending or existing instantiations in the same Ada library into which the actual generic body is compiled (or recompiled), do not need this option. Such pending or existing instantiations are automatically compiled (or recompiled) when the actual generic body is compiled into the same Ada library.  
  
Warning: Compilation (or recompilation) of instantiations either automatically or by using this option only affects instantiations stored as separate units in the Ada library (see `-u`). Existing instantiations which are "inline" in another unit are not automatically compiled or recompiled by using this option. Units containing such instantiations must be explicitly recompiled by the user if the actual generic body is recompiled.
- `-k` Cause the compiler to save an internal representation of the source in the Ada library for use by the Ada cross referencer `ada_xref(1)`. By default, the internal representation is not saved.
- `-lx` Cause the linker to search the HP-UX library named either `/lib/llbx.a` (tried first) or `/usr/lib/llbx.a` (see `ld(1)`).
- `-m <nnn>` The supplied number is the size in Kbytes to be allocated at compile time to manipulate library information. The range is 500 to 32767. The default is 500. The default size should work in almost all cases. In some extreme cases involving very large programs, increasing this value will improve compilation time. Also, if the value is too small, `STORAGE_ERROR` can be raised.
- `-n` Cause the output file from the linker to be marked as shareable (see `-N`). For details refer to `chr(1)` and `ld(1)`.
- `-o outfile` Name the output file from the linker `outfile` instead of `a.out`. In addition, if used with the `-c` option, name the object file output by the binder `outfile.o` instead of `a.out.o`. If debugging is enabled (with `-d` or `-D`), name the debug information file output by the binder `outfile.cul` instead of `a.out.cul`.  
  
The object file output by the binder is deleted if `-c` is not specified.
- `-q` Cause the output file from the linker to be marked as demand loadable (see `-Q`). For details refer to `chr(1)` and `ld(1)`.
- `-r <nnn>` Set listing line length to `<nnn>` (legal range 60..255, default 79). This option applies to the listing produced by both the compiler and the binder (see `-B`, `-L` and `-W b,L`).
- `-s` Cause the output of the linker to be stripped of symbol table information (see `ld(1)` and `strip(1)`).

- t *c,name*** Substitute or insert subprocess *c* with *name* where *c* is one or more of a set of identifiers indicating the subprocess(es). This option works in two modes: 1) if *c* is a single identifier, *name* represents the full path name of the new subprocess; 2) if *c* is a set of (more than one) identifiers, *name* represents a prefix to which the standard suffixes are concatenated to construct the full path name of the new subprocesses.
- The possible values of *c* are the following:
- b* binder body (standard suffix is *adabind*)
  - c* compiler body (standard suffix is *adacomp*)
  - 0* same as *c*
  - l* linker (standard suffix is *ld*)
- u** Cause instantiations of generic program unit bodies to be stored as separate units in the Ada library (see -l).
- If -u is not specified, and the actual generic body has already been compiled when an instantiation of the body is compiled, the body generated by the instantiation is stored "inline" in the same unit as its declaration.
- If -u is specified, or the actual generic body has not already been compiled when an instantiation of the body is compiled, the body generated by the instantiation is stored as a separate unit in the Ada library.
- The -u option may be needed if a large number of generic instantiations within a given unit result in the overflow of a compiler internal table.
- Specifying -u reduces the amount of table space needed, permitting the compiler to complete. However it also increases the number of units used within the Ada library, as well as introduces a small amount of overhead at execution time, in units which instantiate generics.
- v** Enable verbose mode, producing a step-by-step description of the compilation, binding, and linking process on standard error.
- w** Suppress warning messages.
- x** Perform syntactic checking only. The *libraryname* argument must be supplied, although the Ada library is not modified.
- B** Causes the compiler to produce a compilation listing, suppressing page headers and the error summary at the end of the compilation listing. This is useful when comparing a compilation listing with a previous compilation listing of the same program, without the page headers causing mismatches. This option can not be specified in conjunction the -L option.
- C** Only generate checks for stack overflow. Use of this option may result in erroneous (in the Ada sense) program behavior. In addition, some checks (such as those automatically provided by hardware) might not be suppressed. See the Users Guide for more information.
- D** Cause the compiler to store additional information in the Ada library for the units being compiled, for use by the Ada debugger (see *ada.probe(1)*). In addition to saving information required for debugging, an internal representation of the actual source is saved. This permits accurate source level debugging at the expense of a larger Ada library if the actual source file changes after it is compiled. (see -d.) By default, neither debug information nor source information is stored.

Cause the binder to produce a debug information file for the program being bound so that the resulting program can be manipulated by the Ada debugger. The debug information file name is the executable program file name with *.cul* appended. If the debug information file name would be truncated by the file system on which it would be created, an error will be reported.

Only sources compiled with the *-d* or *-D* option contribute information to the debug information file produced by the binder

- G Generate code but do not update the library. This is primarily intended to allow one to get an assembly listing (with *-S*) without changing the library. The *libraryname* argument must be supplied, although the Ada library is not modified.
- I Suppress all inlining. No procedures or functions are expanded inline and pragma *inline* is ignored. This also prevents units compiled in the future (without this option in effect) from inlining any units compiled with this option in effect.
- L Write a program listing with error diagnostics to standard output. This option can not be specified in conjunction with the *-B* option.
- M *<main>* Invoke the binder after all source files named in the command line (if any) have been successfully compiled. The argument *<main>* specifies the entry point of the Ada program; *<main>* must be the name of a parameterless Ada library level procedure.  
  
The library level procedure *<main>* must have been successfully compiled into (or linked into) the named Ada library, either by this invocation of *ada* or by a previous invocation of *ada* (or *ada.umgr(UTIL)*).  
  
The binder produces an object file named *a.out.o* (unless *-o* is used to specify an alternate name), only if the option *-c* is also specified. The object file produced by the binder is deleted unless the option *-c* is specified. Note that the alternate name is truncated, if necessary, prior to appending *.o*.
- N Cause the output file from the linker to be marked as unshareable (see *-n*). For details refer to *char(1)* and *ld(1)*.
- O Invoke the optimizer with full optimizations. See the description of *+O* under the **DEPENDENCIES** section for more information.
- P *<nnn>* Set listing page length to *<nnn>* lines (legal range 10..32767 or 0 to indicate no page breaks, default 66). This length is the total number of lines listed per listing page. It includes the heading, header and trailer blank lines, listed program lines, and error message lines. This option applies to the listing produced by both the compiler and the binder (see *-L* and *-W b,-L*).
- Q Cause the output file from the linker to be marked as not demand loadable (see *-q*). For details refer to *char(1)* and *ld(1)*.
- R Suppress all runtime checks. However, some checks (such as those automatically provided by hardware) might not be suppressed. Use of this option may result in erroneous (in the Ada sense) program behavior.
- S Write an assembly listing of the code generated to standard output. This output is not in a form suitable for processing with *as(1)*.
- W *c,arg1[,arg2,...,argN]*  
Cause *arg1* through *argN* to be handed off to subprocess *c*. The *argi* are of the form *-argoption[,argvalue]*, where *argoption* is the name of an option recognized by the subprocess and *argvalue* is a separate argument to *argoption* where necessary.

The values that *c* can assume are those listed under the *-t* option as well as *d* (driver program).

For example, the specification to pass the *-r* (preserve relocation information) option to the linker would be:

*-W l,-r*

For example, the following:

*-W b,-m,10,-s,2*

sends the options *-m 10 -s 2* to the binder. Note that all the binder options can be supplied with one *-W*, (more than one *-W* can also be used) and that any embedded spaces must be replaced with commas. Note that *-W b* is the only way to specify binder options.

The *-W d* option specification allows additional implementation-specific options to be recognized and passed through the compiler driver to the appropriate subprocess. For example,

*-W d,-O,eo*

sends the option *-O eo* to the driver, which sends it to the compiler so that the *e* and *o* optimizations are performed. Furthermore, a shorthand notation for this mechanism can be used by prepending the option with *+*; as follows:

*+O eo*

This is equivalent to *-W d,-O,eo*. Note that for simplicity this shorthand is applied to each implementation-specific option individually, and that the *argvalue* (if any) is separated from the shorthand *argoption* with white space instead of a comma.

*-X*

Perform syntactic and semantic checking. The *libraryname* argument must be supplied, although the Ada library is not modified.

#### Binder Options

The following options can be passed to the binder using *-W b,...*:

- W b,-b* At execution time, interactive input blocks if data is not available. All tasks are suspended if input data is not available. This option is the default if the program contains no tasks (see *-W b,-B*).
- W b,-k* Keep uncalled subprograms when binding. The default is to remove them.
- W b,-m,<nnn>* Ada/300: Set the initial program stack size to *<nnn>* units of 1024 bytes (legal range 1..32767, default 10 units = 10 \* 1024 bytes = 10240 bytes).  
Ada/800: Set the maximum stack limit of the program stack to *<nnn>* units of 1024 bytes (legal range 512..32767, defaults to a system-defined limit).
- W b,-s,<nnn>* Cause round-robin scheduling to be used for tasking programs. Set the time slice to *<nnn>* tens of milliseconds (legal range 1..32767 or 0 to turn off time slicing). By default, round-robin scheduling is enabled with a time slice of 1 second (*<nnn>* = 100).  
Ada/300: The time slice granularity is 20 milliseconds (*<nnn>* = 2).  
Ada/800: The time slice granularity is 10 milliseconds (*<nnn>* = 1).
- W b,-t,<nnn>* Set task stack size of created tasks to *<nnn>* units of 1024 bytes.  
Set the initial (and maximum) task stack size (legal range 1..32767, default 8 units = 8 \* 1024 bytes = 8192 bytes).

- W b,-w      Suppress warning messages.
- W b,-x      Perform consistency checks without producing an object file and suppress linking. The -W b,-L option can be used to obtain binder listing information when this option is specified (see -W b,-L below).
- W b,-B      At execution time, interactive input does not block if data is not available. Only the task(s) doing interactive input are suspended if input data is not available. This option is the default if the program contains tasks (see -W b,-b).
- W b,-L      Write a binder listing with warning/error diagnostics to standard error.
- W b,-T      Suppress procedure traceback in response to runtime errors and unhandled exceptions.

### Locks

To ensure the integrity of their internal data structures, Ada libraries and families are locked for the duration of operations that are performed on them. Normally Ada families are locked for only a short time when libraries within them are manipulated. However, multiple Ada libraries might need to be locked for longer periods during a single operation. If more than one library is locked, *ada* places an exclusive lock on one library, so it can be updated, and a shared lock on the other(s), so that they can remain open for read-only purposes.

An Ada family or library locked for updating cannot be accessed in any way by any part of the Ada compilation system except by the part that holds the lock. An Ada family or library locked for reading can be accessed by any part of the Ada compilation system desiring to read from the Ada family or library.

If *ada* cannot obtain a lock after a suitable number of retries, it displays an informational message and terminates.

Under some circumstances, an Ada family or Ada library might be locked, but the locking program(s) might have terminated (for example, due to system crash or network failure). If you determine that the Ada family or Ada library is locked but should not be locked, you may remove the lock.

Use *ada.unlock(1)* to unlock an Ada library and *ada.funlock(1)* to unlock an Ada family. However, unlocking should be done with care. If an Ada family or Ada library is actually locked by a tool, unlocking it will permit access by other tools that might find the contents invalid or that might damage the Ada family or Ada library.

### DIAGNOSTICS

The diagnostics produced by *ada* are intended to be self-explanatory. Occasional messages might be produced by the linker.

If a program listing (-B or -L) and/or generated code listing (-S) is requested from the compiler, this listing as well as compiler error messages are written to standard output.

If neither a program listing nor a generated code listing is requested from the compiler, erroneous source lines and compiler error messages are written to standard error.

If a binder listing is requested from the binder (with -W b,-L), the binder listing as well as binder error messages are written to standard error.

If a binder listing is not requested from the binder, binder error messages are written to standard error.

Errors detected during command line processing or during scheduling of the compiler, binder, or linker, are written to standard error. If any compiler, binder, or linker errors occur, *ada* writes a one-line summary to standard error immediately before terminating.

## WARNINGS

Options not recognized by *ada* are not passed on to the linker. The option *-W Larg* can be used to pass such options to the linker.

*Ada* does not generate an error or warning if both optimization and debugging are requested. However, *ada.probe* is only capable of limited debugging of optimized code. Certain *ada.probe* commands may give misleading or unexpected results. For example, object values may be stored in registers; therefore the value displayed from memory may be incorrect. For this reason, the ability to examine or modify objects and expressions may be impaired. Dead code elimination or code motion may affect single step execution or prevent breakpoints from being set on specific source lines.

## DEPENDENCIES

## Series 300

The binder option *-W b,-T* also causes traceback tables to be excluded from final executable file.

The following options are specific to the Series 300:

**+O *what*** Selectively invoke optimizations. The *what* argument must be specified, and indicates which optimizations should be performed. Note that the option *-O* is equivalent to *+O eloE*.

The *what* argument can be a combination of the letters *e*, *l*, *o*, *p*, *E*, and *P*. Either *e* or *p*, but not both, can be specified. Similarly, either *E* or *P*, but not both, can be specified. All other combinations are permitted; however, at most one of each letter can be specified.

**e** Same as *p* (below).

**l** Permit procedures and functions not declared with *pragma inline* to be expanded inline at the compiler's discretion. Only procedures and functions in the current source file are considered.

Procedures and functions declared with *pragma inline* are always considered candidates for inline expansion unless *-I* is specified; this optimization only causes additional procedures and functions to be considered.

**o** Peephole optimizations are performed on the final object code.

**p** Optimizations are performed to remove unnecessary checks, optimize loops, and remove dead code.

**E** Same as *P* (below).

**P** Optimizations are performed on common subexpressions and register allocation.

**+h *<type>*** Bind/link the program to use the specified *<type>* of hardware floating point assist for user code floating point operations (see *+H*). The two *<type>*'s currently supported are 68881 (the 68881 math coprocessor) and 68882 (the 68882 math coprocessor). The code generated is the same for either *<type>*. This is the default if the host system provides a 68881 or a 68882 coprocessor.

**+i *<type>*** Compile the program to inline the specified *<type>* of hardware floating point assist for user code floating point operations (see *+I*). The two *<type>*'s currently supported are 68881 (the 68881 math coprocessor) and



68882 (the 68882 math coprocessor). The code generated is the same for either *<type>*. This is the default if the host system provides a 68881 or a 68882 coprocessor.

- +H** Bind/link the program to use software floating point routines for user code floating point operations (see **+h**). This is the default if the host system does not provide a 68881 or 68882 coprocessor.
- +I** Compile the program to make calls to a math library for user code floating point operations (see **+i**). The **+h** option is then used at bind/link time to specify whether hardware or software is used for floating point operations. This is the default if the host system does not provide a 68881 or 68882 coprocessor.

Unlike other Series 300 compilers, it is not possible to link-only using the *ada(1)* command. If separate linking is desired, use the *ld(1)* command.

A successful bind produces a (non-executable) *.o* file. The *.o* file is normally deleted unless the compiler option **-c** is specified.

#### Series 800

The binder option **-W b,-T** also causes traceback tables to be excluded from final executable file.

The following options are specific to the Series 800:

- +O *what*** Selectively invoke optimizations. The *what* argument must be specified, and indicates which optimizations should be performed. Note that the option **-O** is equivalent to **+O e1E**.

The *what* argument can be a combination of the letters *e*, *i*, *l*, *p*, *E*, and *P*. Either *e* or *p*, but not both, can be specified. Similarly, either *E* or *P*, but not both, can be specified. All other combinations are permitted; however, at most one of each letter can be specified.

- e** Same as *p* (below).
- i** Permit procedures and functions not declared with `pragma inline` to be expanded inline at the compiler's discretion. Only procedures and functions in the current source file are considered.  
Procedures and functions declared with `pragma inline` are always considered candidates for inline expansion unless **-I** is specified; this optimization only causes additional procedures and functions to be considered.
- l** The code generator performs level 1 optimizations.
- p** Optimizations are performed to remove unnecessary checks, optimize loops, and remove dead code.
- E** Same as *P* (below).
- P** Optimizations are performed on common subexpressions and register allocation.

- +T** Suppress the generation of traceback information at compile time. In addition to suppressing traceback at run time, this also reduces the size of the object file in the *ada* library.

Unlike other Series 800 compilers, it is not possible to link-only using the *ada(1)* command. If separate linking is desired, use the *ld(1)* command.

A successful bind produces a (non-executable) *.o* file. The *.o* file is normally deleted unless the compiler option *-c* is specified.

## AUTHOR

*Ada* was developed by HP and Alsys.

## FILES

file.ad?	input file (Ada source file).
libraryname	user Ada library (created using <i>ada.mklib(1)</i> ) in which compiled units are placed by a successful compilation and from which the binder extracts the units necessary to build a relocatable file for <i>ld(1)</i> . Temporary files generated by the compiler are also created in this directory and are automatically deleted on successful completion. Users are strongly discouraged from placing any additional files in Ada library directories. User files in Ada libraries are subject to damage by or may interfere with proper operation of <i>ada</i> and related tools.
file.o	binder-generated object file or user-specified object file relocated at link time.
a.out	linked executable output file.
file.cui	binder-generated debug information file.
\$ADA_PATH/ada	Ada compilation control program.
\$ADA_PATH/adacomp	Ada compiler.
\$ADA_PATH/adabind	Ada binder.
\$ADA_PATH/ada_environ	Ada environment description file.
\$ADA_PATH/adaargu	Ada argument formatter.
\$ADA_PATH/alternate	Ada predefined library (sequential version).
\$ADA_PATH/installation	Ada installation family.
\$ADA_PATH/public	Ada public family.
\$ADA_PATH/err_tpl	Ada compiler/binder error message files.
\$ADA_PATH/predeflib	Ada predefined library, tasking version.
\$ADA_PATH/libada020.a	Ada run-time HP-UX library (MC68020).
\$ADA_PATH/libada881.a	Ada run-time HP-UX library (MC68881).
/lib/crt0.o	C run-time startup.
/lib/libc.a	HP-UX C library.
/lib/libm.a	HP-UX math library.

## SEE ALSO

*ada.fmgr(1)*, *ada.format(1)*, *ada.funiock(1)*, *ada.lmgr(1)*, *ada.lsfam(1)*, *ada.lslib(1)*, *ada.lake(1)*, *ada.mkfam(1)*, *ada.mklib(1)*, *ada.mvfam(1)*, *ada.mvlib(1)*, *ada.probe(1)*, *ada.project(1)*, *ada.rmfam(1)*, *ada.rmlib(1)*, *ada.umgr(1)*, *ada.unlock(1)*, *ada.xref(1)*,

*Ada User's Guide*,

*Ada Tools Manual*,

*Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A),

*Reference Manual for the Ada Programming Language*, Appendix F (Series 300). *Reference Manual for the Ada Programming Language*, Appendix F (Series 800).

## EXTERNAL INFLUENCES

International Code Set Support

Single-byte character code sets are supported within file names.